Analysis and Optimization of Deep Counterfactual Value Networks

Analyse und Optimierung von Tiefen Counterfactual Value Netzwerken
Bachelor-Thesis von Patryk Hopner
April 2018

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Knowledge Engineering
Prof. Dr. Johannes Fürnkranz

Analysis and Optimization of Deep Counterfactual Value Networks
Analyse und Optimierung von Tiefen Counterfactual Value Netzwerken

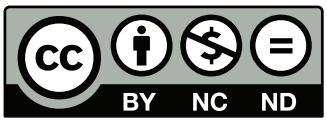
Vorgelegte Bachelor-Thesis von Patryk Hopner

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Eneldo Loza Mencía

Tag der Einreichung:

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-12345
URL: http://tuprints.ulb.tu-darmstadt.de/1234

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de



Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung 2.0 Deutschland
http://creativecommons.org/licenses/by-nc-nd/2.0/de/

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Patryk Hopner, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Datum / Date: Unterschrift / Signature:

_____ _____

Contents

1 Introduction

Poker has been an interesting subject for many researchers in the field of machine learning and artificial intelligence over the past decades. Unlike games like chess or checkers it involves imperfect information, making it unsolvable using traditional game solving techniques.
In a perfect information game, such as checkers, solutions for situations in the endgame can be computed independently and the board configuration can be marked as either a winning or losing state for a given player. Those solutions can then be combined to a solution of the full game, which is not possible in a game like poker.
In a perfect information game all players are aware of the current game state, which leads to the fact, that the way in which the players arrived at the current state can be ignored. In poker past actions can not be ignored, as the players can not see the opponent cards and past actions change the probability of an opponent holding a certain hand. This means, that the game can not be simply divided into sub-problems, but instead the solution for the whole game tree has to be computed at once, which results in high memory requirements for human scale imperfect information games.
In 2015 a solution was found for the game of Fixed-Limit Hold'em, the more popular No-Limit variant however has a game tree several orders of magnitude bigger than Fixed-Limit and remains unsolved to this day.
For many years the state of the art approach for creating strong poker agents for the game of No-Limit involved computing an approximate Nash equilibrium in a smaller, abstract game, using algorithms like counterfactual regret minimization and then mapping the results back to situations in the real game.
Those abstracted games are however several orders of magnitude smaller than the actual game tree of No-Limit poker, which means, that the poker agent has to treat many strategically different situations as if they were the same.
Before Fixed-Limit was solved, there existed abstracted solutions of the game, which, while not being perfect solutions of the full game, were able to defeat human professional poker players. This was not yet achieved for the game of No-Limit using traditional abstraction based methods.
The main reason for the better performance of abstraction based Fixed-Limit poker bots was the fact, that the full game was only about 10 times bigger than the abstraction. No-Limit has a dramatically bigger game tree and abstraction based approaches do not seem to scale well enough for No-Limit, which creates the need for new approaches.
Recently a work was published, combining ideas from traditional poker solving algorithms with ideas from perfect information games, creating the strong poker agent called DeepStack. DeepStack solved many of the problems associated with abstraction based techniques and was also able to beat professional poker players with statistical significance.
The algorithm does not need to pre-compute a solution for the whole game tree, instead it computes a solution during game play. In order to make solving the game during game play computationally feasible, DeepStack does not traverse the whole game tree, instead it uses an estimator for values of future states. For that purpose a deep neural network was created, using several million solutions of poker sub-games as training data, which were solved using traditional poker solving algorithms. It has been proven, that the algorithm used by DeepStack approaches a Nash equilibrium as the quality of the predictions of its neural network improves.

1.1 Contributions

This thesis will focus on the neural networks, which DeepStack uses for the prediction of values of future states, called deep counterfactual value networks.
It has been proven, that, given a counterfactual value network with perfect accuracy, the solution produced by DeepStack will converges to a Nash equilibrium of the game. This means on the other hand, that wrong predictions of the network can result in a bad solution, we will therefore analyze potential problems in the

design of DeepStack's networks and try to improve upon them.
For the training of the network, training examples have to be created by solving sub-trees of the game, using traditional solving algorithms. Those training examples have to cover as many different situations as possible, in order for the neural network to generalize well to unseen situations. We will analyze the way in which the sub-tree situations were created and the impact on the accuracy of the network. As solving the sub-games is computationally expensive and we need a big data set in order for the network to learn a good model, we will also show ways of speeding up the process.
Besides the quality and quantity of training examples, one of the most important factors for the accuracy of a neural network is the way the inputs and outputs are encoded. DeepStack uses an encoding based on traditional poker abstractions, which can potentially lead to problems. We will analyze the encoding and compare it to other traditional methods. We will then present a new, non domain specific encoding scheme, which can also be used in older imperfect information game solving algorithms.

1.2 Thesis Structure

The structure of this thesis is as follows: chapter 2 provides background information for the game of poker as well as the rules of the two popular poker variants No-Limit Hold'em and Fixed-Limit Hold'em. Chapter 3 provides the theoretical background necessary for solving imperfect information games such as poker as well as descriptions of traditional techniques, which are then contrasted to the technique used by DeepStack.
Chapter 4 provides details for a fast implementation of counterfactual regret minimization, which is necessary for the creation of training data for deep counterfactual value networks.
Chapter 5 describes our version of counterfactual value networks and analyzes several potential problems of deep counterfactual value networks.
Chapter 6 tries to find new ways of encoding the inputs and outputs of deep counterfactual value networks based on traditional card abstraction techniques and finally chapter 7 describes a new card abstraction technique derived from the training data used by the counterfactual value network.

2 Poker

The term poker describes a family of card games in which players receive one or more private cards, which are unknown to the opponents. Players are then betting on whose cards have the highest rank, according to the rules of the game. There are many variants of the game and they can vary in the number of private cards, the number of players, the number of publicly visible cards and the number and sizes of possible bets. This thesis will focus on the most popular poker variant to date, **No-Limit** (NL) **Hold'em**. The game of **Fixed-Limit** (FL) Hold'em, which only allows bets of a fixed size, will also be introduced, as it is relevant to related work, especially the CFR+ algorithm, which will be described in a later chapter. For both variants we will focus on the 2 player variant, called **Heads-Up** (HU).

2.1 No-Limit Hold'em

No-Limit Hold'em is played with a standard deck of 52 cards, with the 2 having the lowest rank and the Ace having the highest. Cards can have one of 4 **suits**: **hearts** (♡, h), **clubs** (♣, c), **diamonds** (♢, d) and **spades** (♠, s), the suit of a card does not influence its strength.
The players start the game with a certain amount of chips, called their **stacks**. Players can never invest more than their stack during a game. There are 4 rounds in a game of NL Hold'em, called **preflop**, **flop**, **turn** and **river**. At the start of a game players are required to make forced bets, called **blinds**, with the **big blind** being twice the size of the **small blind**. Players are then dealt 2 cards face down, called **private cards** or **hole cards**. After the players received their cards the player who posted the small blind starts the betting.
The possible actions are: **fold**, which means giving up the hand and the chips invested so far, **check**, which means passing the turn without investing money, **call**, which means matching the opponent's bet and **raise**, forcing the opponent to either fold his hand or invest more chips. Raises can have varying sizes, a raise which commits all of the chips in a player's stack is called an **all-in**.
If none of the players folded during the first round of the game, the second round starts and 3 **public** or **community cards** are revealed, which form the **public board** and can be combined with each player's private cards, forming a **hand**. On the third and fourth round one additional public card each is revealed. If none of the players folded his hand, players reveal their hole cards in a **showdown** and the highest ranked hand wins the chips in the pot. After a game is over, players rotate seats and the next game is played.

Table 1: Hand ranks, highest to lowest

Hand	**Definition**	**Example**
Royal Flush	5 cards with the same suit and consecutive ranks, with an Ace as highest card	A♡ K♡ Q♡ J♡ T♡
Straight Flush	5 cards with the same suit and consecutive ranks	Q♡ J♡ T♡ 8♡ 9♡
Four of a Kind	4 cards of the same rank	J♣ J♠ J♢ J♡ 7♣
Full House	3 cards with the same rank and a pair	K♠ K♡ K♢ 8♠ 8♢
Flush	5 cards with the same suit	Q♣ 9♣ 7♣ 2♣ 3♣
Three of a Kind	3 cards with the same rank	J♠ J♢ J♡ 2♣ 3♣
Two Pair	2 pairs	J♣J♡9♠9♢2♣
Pair	2 cards with the same rank	J♣ J♠ 9♠ 6♣ 5♢
High Card	None of the above	A♣ T♠ 6♢ 7♣ 2♣

2.2 Fixed-Limit Hold'em

Fixed-Limit Hold'em Heads-Up is the smallest variant of poker which is being played by a significant number of people. To many this makes it a more interesting area of research than smaller poker toy-games, such as Kuhn Poker [Kuhn, 1950], which are only of interest to researchers. While the popularity of FL

Hold'em has declined over the past years, with many players choosing the No-Limit variant instead, it is still played, both in casinos and online.
In 2015 FL Hold'em HU has been essentially solved by the Computer Poker Research Group at the University of Alberta [O. Tammelin, N. Burch, M. Johanson, and M. Bowling, 2015], meaning that an approximated Nash equilibrium has been computed, which can not be beaten by more than 0.0001 big blinds per game. This makes it impossible to distinguish it, with statistical significance, from a perfect Nash equilibrium during the lifespan of a human.
After this achievement many researchers chose to move to more challenging games like NL Hold'em or to the multiplayer variant of Fixed-Limit. FL Hold'em is nevertheless a good way of testing ideas which can be ultimately used for an NL Hold'em agent and many of the techniques presented in this thesis have been developed with the game of FL Hold'em in mind. It will help us better understand the progress in the domain of artificial intelligence for poker and by contrasting it to NL Hold'em also the new challenges that we face.
The rules of FL Hold'em only differ from NL Hold'em in the number and size of possible bets. While in NL Hold'em the size of a bet or raise is variable, it is determined by the rules of the game in FL Hold'em. The size of bets and raises during the first 2 rounds is equal to the size of the big blind, the size during the last 2 rounds is twice the size of the big blind. The number of raises on each betting round is also limited, players can make only up to 4 raises, after which the opponent can only fold or call, but not re-raise.

3 Background and Related Work

This chapter will provide the necessary background information needed in order to understand later chapters. It will start by introducing game theoretical concepts, such as the concept of a Nash equilibrium, a best response and exploitability.
We will then present state of the art methods for the computation of a Nash equilibrium, such as Counterfactual Regret Minimization and CFR+.
This will be followed by abstraction techniques, which are needed to solve a game as big as NL Hold'em HU with Counterfactual Regret Minimization. After a presentation of popular abstraction techniques their benefits and weaknesses will be discussed.
We will end by analyzing DeepStack, a new algorithm for solving imperfect information games such as poker, which was the inspiration for this thesis. After an introduction to the algorithm, the ways in which DeepStack attempts to solve some of the problems with earlier algorithms, which were described in this chapter, will be discussed.

3.1 Nash Equilibrium

As described in [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007] poker can be modeled as an **extensive game**, in which player actions and chance events form a game tree, similar to chess or checkers. The nodes of the tree correspond to **histories** of actions $h \in H$. Each non-terminal history has an acting player $p(h) \in P$ associated with it, who must select an action $a \in A(h)$. A terminal history $z \in Z \subset H$ has associated utilities $u_i(z)$ for player i. In the case of Hold'em terminal histories correspond to showdowns and folds.
Since poker is an **imperfect information** game, meaning that some information is hidden from the acting player, the concept of **information sets** is introduced. An information set $I \in I_i$ of player i is a set of histories or states, which can not be distinguished from that player's perspective. In the case of Hold'em a player can distinguish states by the betting history, by the private cards he is holding and by public cards, but not by the cards his opponent is holding.
A player's strategy $\sigma_i \in \Sigma_i$ is a function which assigns a probability to each legal action a player can take in a given information set. As all states in one information set are indistinguishable from each other, a player must use the same strategy in all the states in that information set, intuitively this means he must choose a strategy without knowing the opponent's hand distribution. A strategy profile σ is a function which assigns a strategy to each player involved in the game.
A **Nash equilibrium** [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007] [J. F. Nash, 1951] is a strategy profile in which no player can increase his expectation by unilaterally deviating from the strategy profile. As we are focusing on the two player case, this can be formally expressed by

$$u_1(\sigma) \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2) \qquad u_2(\sigma) \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2) \tag{1}$$

A **best response** strategy $br_i(\sigma)$ for player i is a strategy maximizing the expectation against a given strategy profile, the corresponding best response value $b_i(\sigma)$ is the utility the best response strategy can achieve. A **zero sum** game is a game in which the expectations of all players sum to zero. In a two player zero sum game, such as NL Hold'em HU, a best response to a Nash equilibrium strategy can not achieve an expectation higher than zero. This means that there exists no strategy, that can win against a strategy, which is part of a Nash equilibrium strategy profile.
A Nash equilibrium can be approximated, a strategy profile which loses by at most ϵ to a best response is called an **ϵ-Nash equilibrium**.

$$u_1(\sigma) + \epsilon \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2) \qquad u_2(\sigma) + \epsilon \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2) \tag{2}$$

In a two player zero sum game we can define the **exploitability** of σ, ϵ_σ as $\frac{(b_1(\sigma_2)+b_2(\sigma_1))}{2}$ which describes the maximum a strategy profile can lose on average, if players rotate seats after every game.
We define the attempt of a player to increase his expectation against a strategy, by deviating from a Nash equilibrium, as **exploitation**, regardless if he uses a best response strategy or a sub-optimal response.

3.2 Counterfactual Regret Minimisation

Counterfactual Regret Minimisation (CFR) [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007] describes a family of regret minimizing algorithms used for solving imperfect information games. The algorithms have a linear time and space complexity in the number of information sets [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007] and the first version, usually referred to simply as **CFR** or **Vanilla CFR** was first introduced in 2007 by researchers at the University of Alberta.
Since then many versions of the base algorithm have been developed, including Monte Carlo methods such as Chance Sampling [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007], Public Chance Sampling [M. Johanson, N. Bard, M. Lanctot, R. Gibson, and M. Bowling, 2012] and Pure CFR [R. Gibson, 2014].
In [O. Tammelin, 2014] a new enumerative CFR algorithm, called **CFR+** was introduced. CFR+ will be described later in this section.
CFR algorithms are the state of the art algorithms for finding approximate Nash equilibria in imperfect information games and were the basis for the creation of many strong poker bots such as the FL Hold'em HU bots Hyperborean [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007] and Polaris [M. Johanson, 2007] and the NL Hold'em HU bots Hyperborean NL [D.P. Schnizlein, 2009], Tartanian [A. Gilpin, T. Sandholm and T.B. Sorensen, 2008] and Libratus [N. Brown and T. Sandholm, 2017].
In the next sections CFR and CFR+ will be described in detail.

3.2.1 CFR

CFR is based on self play and tries to find an approximate Nash equilibrium by minimizing average **counterfactual regret**, which has been shown to also minimize overall average regret [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007].
Following the definitions in [M. Johanson, N. Bard, M. Lanctot, R. Gibson, and M. Bowling, 2012] we first define Z_I as the set of terminal histories z passing through information set I and $z[I]$ as the prefix of z contained in I. Furthermore $\pi^\sigma_{-i}(h)$ is the probability of reaching history h, given strategy profile σ, if player i always tried to reach the information set, in other words it ignores player i's contribution to the probability. Finally $\pi^\sigma(h, h')$ is the probability of reaching history h' given strategy profile σ and assuming history h was reached.
The **counterfactual value** v_i of player i at information set I and given strategy profile σ can then be defined as in [M. Johanson, N. Bard, M. Lanctot, R. Gibson, and M. Bowling, 2012] by

$$v_i(\sigma, I) = \sum_{z \in Z_I} u_i(z) \pi^\sigma_{-i}(z[I]) \pi^\sigma(z[I], z)$$

with $u_i(z)$ being the utility for player i at the terminal history z, as described in the last section.

Counterfactual regrets for action a in information set I and iteration T are defined as

$$R_i^T(I,a) = \begin{cases} v_i(\sigma^T_{I\to a}, I) - v_i(\sigma^T, I) & T = 1 \\ R_i^{T-1}(I,a) + v_i(\sigma^T_{I\to a}, I) - v_i(\sigma^T, I) & T > 1 \end{cases}$$

with $\sigma_{I\to a}$ being the same as the strategy profile σ, except that action a is always played in information set I.
The counterfactual regret of an action a in information set I measures the difference between the expectation of using the current strategy profile σ, compared to a strategy profile which always plays the action a in this information set. Intuitively this measures how much a player "regrets" not always playing action a in this information set, hence the name.
The strategy for the next iteration is determined by the average regrets of each action, let us define $R_i^{+,T}(I,a) = \max(R_i^T, 0)$, then the new strategy is

$$\sigma^{T+1} = \begin{cases} \frac{R_i^{+,T}(I,a)}{\sum_{a'\in A(I)} R_i^{+,T}(I,a')} & \text{if the denominator is positive} \\ \frac{1}{|A(I)|} & \text{otherwise} \end{cases}$$

Stronger actions, which have a high counterfactual value, will also have high average regrets, making the current strategy of the agent play strong actions more often. Actions with a non positive average regret will not be played until the average regret becomes positive. Although the current strategy, which is proportional to the positive average regrets, can produce a good strategy, it is in general not guaranteed to converge to a Nash equilibrium [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007]. For each action a in information set I the average strategy $\bar{\sigma}_i^T$ for player i at iteration T can be defined as

$$\bar{\sigma}_i^T(I)(a) = \frac{1}{T}\sum_{t=1}^{T} \sigma_t(I)(a)$$

with σ_t being the strategy profile in iteration t. It was proven in [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007], that the average strategy computed using CFR is guaranteed to converge to a Nash equilibrium.

3.2.2 CFR+

CFR+ was created by Oskari Tammelin and described in [O. Tammelin, 2014]. It is the algorithm that was used by the University of Alberta Computer Poker Research Group in order to create Cepheus [O. Tammelin, N. Burch, M. Johanson, and M. Bowling, 2015] and essentially solve the game of FL Hold'em HU.
The main difference between Vanilla CFR and CFR+ is the regret matching algorithm used, called **Regret Matching+**, which, using the definitions from the last section, is formally defined in [O. Tammelin, 2014] as

$$R_i^{+,T}(I,a) = \begin{cases} \max\{v_i(\sigma^T_{I\to a}, I) - v_i(\sigma^T, I), 0\} & T = 1 \\ \max\{R_i^{+,T-1}(I,a) + v_i(\sigma^T_{I\to a}, I) - v_i(\sigma^T, I), 0\} & T > 1 \end{cases}$$

The new strategy is produced by:

$$\sigma^{T+1} = \begin{cases} \frac{R_i^{+,T}(I,a)}{\sum_{a'\in A(I)} R_i^{+,T}(I,a')} & \text{if the denominator is positive} \\ \frac{1}{|A(I)|} & \text{otherwise} \end{cases}$$

The difference to the regret matching algorithm used in Vanilla CFR is, that regrets can never become negative. This has multiple advantages, the first being that an action which has a negative regret in Vanilla CFR might not be considered by the algorithm for several iterations, until the regret becomes positive again, while CFR+ will consider the action as soon as it has a positive expectation. This has been empirically shown to improve the convergence rate [O. Tammelin, N. Burch, M. Johanson, and M. Bowling, 2015], which was also confirmed during the writing of this thesis. Figure 1 shows the decrease of exploitability at the start of the turn round for CFR and CFR+. For both algorithms a best response was computed every 200 iterations and the results of 1000 turn situations were averaged.

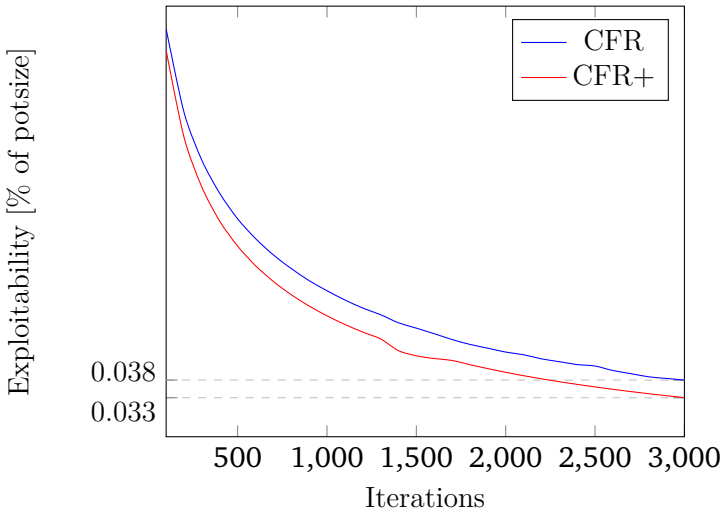


Figure 1: Convergence of CFR and CFR+ averaged over 1000 situations

With any CFR algorithm, only the average strategy is guaranteed to converge to a Nash equilibrium, the current strategy, based on current regrets can be highly exploitable. It has been however observed, that with CFR+ the current strategy often also approaches a very good Nash equilibrium approximation, which is usually not the case with Vanilla CFR. This means, that a very good solution can be found, without storing the average strategy, which lowers the required memory of the algorithm by almost 50%. This can be essential when working with a big game tree and was one of the things making Cepheus possible. In the case of Cepheus the current strategy was exploitable for only 0.0001 big blinds per game, showing the usually good convergence of CFR+ [O. Tammelin, N. Burch, M. Johanson, and M. Bowling, 2015]. The average strategy can however still be used, if memory constrains aren't an issue, as it tends to converge faster than the current strategy. The implementation written for this thesis used the average strategy, as the turn sub games were small enough.
The last advantage is, that because regrets can never become negative, there will be many actions with a regret value of zero. This means that the regrets can be effectively compressed, with observed compression rates of up to 1:13 [O. Tammelin, N. Burch, M. Johanson, and M. Bowling, 2015], which was another technique needed to essentially solve FL Hold'em HU.

3.3 Card Abstraction

Traditional poker solving algorithms like Counterfactual Regret Minimization attempt to find an approximate Nash equilibrium of the game in an offline computation. They have to repeatedly traverse the game tree and store the regret values and strategies for every information set, which will be typically saved in look-up tables and used during play.
While the game of FL Hold'em HU was essentially solved in 2015, without using any form of abstraction, this was only possible after years of research and required 900 CPU years and, even after compressing the

strategy, 11 Terabytes of Memory [M. Bowling, N. Burch, M. Johanson and O. Tammelin, 2015].
The game of No-Limit Hold'em HU, played with a stack size of 200 big blinds, a big blind size of 100$ and allowing only integral bet sizes, which is the standard format of the Annual Computer Poker Competition [ACPC], is a game with $6.3 \cdot 10^{164}$ non-terminal states and $1.39 \cdot 10^{48}$ information sets [M. Johanson, 2013], compared to $3.16 \cdot 10^{17}$ non-terminal states and $3.19 \cdot 10^{14}$ information sets in the case of Fixed-Limit Hold'em HU [M. Bowling, N. Burch, M. Johanson and O. Tammelin, 2015], making it many orders of magnitude too big to be solved on current hardware.
A common approach is to solve a smaller, abstracted version of the game and to map the solution back to the real game during game play. One possible way to create a smaller game is to restrict the number of possible actions a player can take. This is called action abstraction and will be covered in the next chapter, this chapter will describe the technique of **card abstraction**.
Card abstraction is the process of grouping a number of cards together and mapping them to **buckets** [M. Johanson, 2007]. During training cards in the same bucket are considered indistinguishable and only a strategy for each bucket is stored, not a strategy for every individual hand. During game play the strategy of a bucket is then mapped to the actual hand the player is holding.
Besides the usefulness for creating smaller games, which can be solved by a CFR algorithm, card abstractions can also be used to create a feature set for Deep Counterfactual Value Networks [M. Moravcik et al., 2017], which are the focus of this thesis and will be covered in detail in section 3.6.
Card abstractions can be categorized by several properties. The next sections will explain the difference between lossy and lossless as well as perfect and imperfect recall abstractions. We will then give examples of popular card abstractions, some of which we will use to create feature sets for Deep Counterfactual Value Networks later.

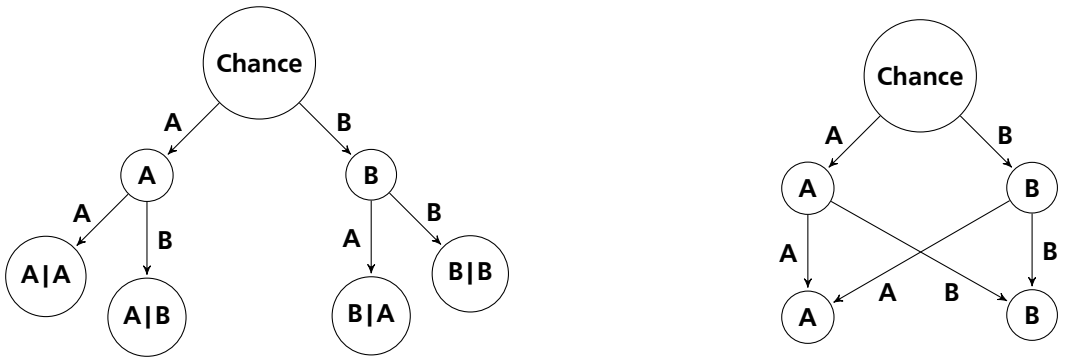
3.3.1 Isomorphisms

Poker hands, which are strategically equivalent and only differ by a rotation of their suits are called **isomorphic**. As an example of that we can consider the first betting round, before any public cards have been revealed, in this situation the hands A♡Q♡, A♠Q♠, A♣Q♣ and A♢Q♢ are all isomorphic to each other.
One of the hands can be arbitrarily chosen as the representative of the group, we call such a representative **canonical**. An algorithm like CFR only needs to compute and store the strategy for each canonical hand. Because every hand is isomorphic to some canonical hand, meaning both hands are strategically equivalent, the strategy of a canonical hand can be used for the non-canonical hand during game play. Because all hands that are grouped together are strategically equivalent, this is a **lossless** abstraction technique, all other techniques presented in this section are **lossy**, meaning that the resulting strategy will potentially perform worse in the real game, than a solution which was computed using the full game tree.
Using isomorphisms can reduce the state space by up to a factor of $4!$ [K. Waugh, M. Zinkevich, M. Bowling and M. Johanson, 2011], which is not yet enough to be able to solve a game as big as NL Hold'em HU, which forces us to use lossy abstraction. Isomorphisms can be used together with lossy abstraction techniques, the bucketing method only needs to consider each canonical hand and not all possible hands.
In [K. Waugh, 2013] Kevin Waugh describes a optimal hand isomorphism algorithm, which was used during the work on this thesis.

3.3.2 Perfect and Imperfect Recall

As described in [M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013] a player has **perfect recall** if he never forgets any action he, his opponent, or chance took. If a player forgets some or all past actions, he is said to have **imperfect recall**. While perfect and imperfect recall are usually not a characteristic of an actual game, as players will generally not forget past information, they are useful concepts for an abstract model of the game.
A **bucket-sequence** is the sequence of buckets to which a player's hand was mapped on each betting round. As an example a hand which was mapped to bucket number 5 in the first betting round and bucket number

12 in the second betting round, has the bucket-sequence 5-12 at that point.
In a perfect recall card abstraction the algorithm will compute a separate strategy for each individual bucket-sequence, while an imperfect recall card abstraction will not distinguish between hands, which are mapped to the same bucket on the current betting round, even if the prefixes of the betting-sequences differ. Because a perfect recall abstraction must store a separate strategy for each bucket-sequence, the number of buckets on early rounds influences the number of possible buckets on later rounds, unlike imperfect recall abstractions. As an example of that let us consider a perfect recall abstraction with 10 buckets on the first round and 10 buckets on the second round. On the second round there exist already 100 possible bucket sequences, therefore 100 strategies for the second round have to be stored. An imperfect recall abstraction does not differentiate between bucket sequences with different prefixes, as long as the hands are mapped to the same bucket on the current round. Therefore an imperfect recall abstraction only has to store 10 strategies on the second round. This also means, that the imperfect recall abstraction could increase the number of buckets on the second round to 100 and use the same amount of memory as the perfect recall abstraction with only 10 buckets, imperfect recall abstractions therefore can forget past information in favor of a finer grained abstraction in the current round. Figure 2 shows an example of a perfect and imperfect recall abstraction with two buckets.



(a) Perfect Recall

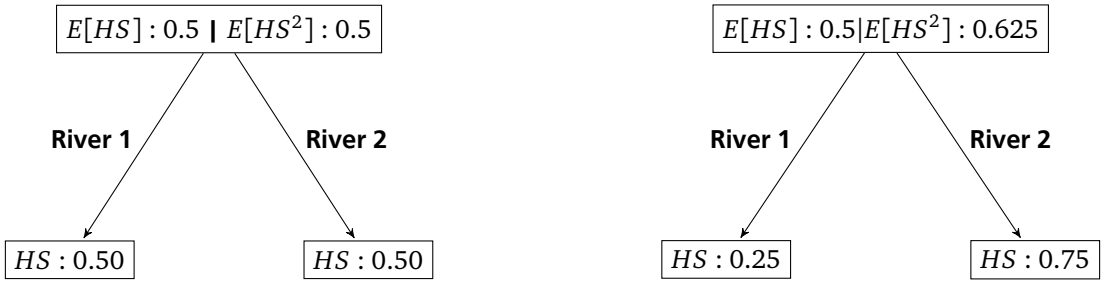
(b) Imperfect Recall

Figure 2: Perfect and Imperfect recall with possible Buckets A and B

While imperfect recall abstractions lose the guarantee of convergence to a Nash equilibrium [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007], it has been shown empirically that they usually outperform perfect recall abstractions of equal size in practice [M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013].

3.3.3 Expectation Based Card Abstractions

As described in [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007], on the last betting round the **hand strength** (HS) value of a hand is the probability of winning against a uniform opponent hand distribution. The **expected hand strength** ($E[HS]$)[M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007] on earlier rounds is calculated by averaging the HS values over all possible card roll outs. A similar metric is the **expected hand strength squared** ($E[HS^2]$) [M. Johanson, 2007] which is obtained by averaging the square of the HS value over all possible card roll outs. While the $E[HS]$ metric only describes the chances of a hand winning at a showdown against a uniform distribution, the $E[HS^2]$ metric tries to take the potential of a hand on different rivers into account. The idea is, that a hand which will be very strong on some rivers and weak on others is generally better, than a hand which will be only mediocre on most rivers. Figure 3 shows an example of two hands with equal $E[HS]$ values but different $E[HS^2]$ values.



(a) Hand 1 (b) Hand 2

Figure 3: Example of 2 hands with equal $E[HS]$ values but different $E[HS^2]$

The $E[HS]$ and $E[HS^2]$ card abstractions use the $E[HS]$ and $E[HS^2]$ values in order to group hands, which are considered to be similar according to the metric, into a bucket.
There are several ways to map hands with given $E[HS]$ / $E[HS^2]$ values to a bucket, including percentile bucketing, which creates equally sized buckets, clustering of hands with an algorithm such as k-Means or by simply grouping hands together, that differ only by a certain threshold in their $E[HS]$ / $E[HS^2]$ values. Both abstractions can be implemented in the perfect recall and imperfect recall variations, with all the benefits and drawbacks of each variation as described in the last section.
The $E[HS]$ card abstraction was used in several works including [D. Billings et al., 2003] and [A. Gilpin and T. Sandholm, 2006], the $E[HS^2]$ card abstraction was first described in [M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, 2007], it was used by Polaris, the first poker bot to ever beat human professional poker players in the game of FL Hold'em HU.

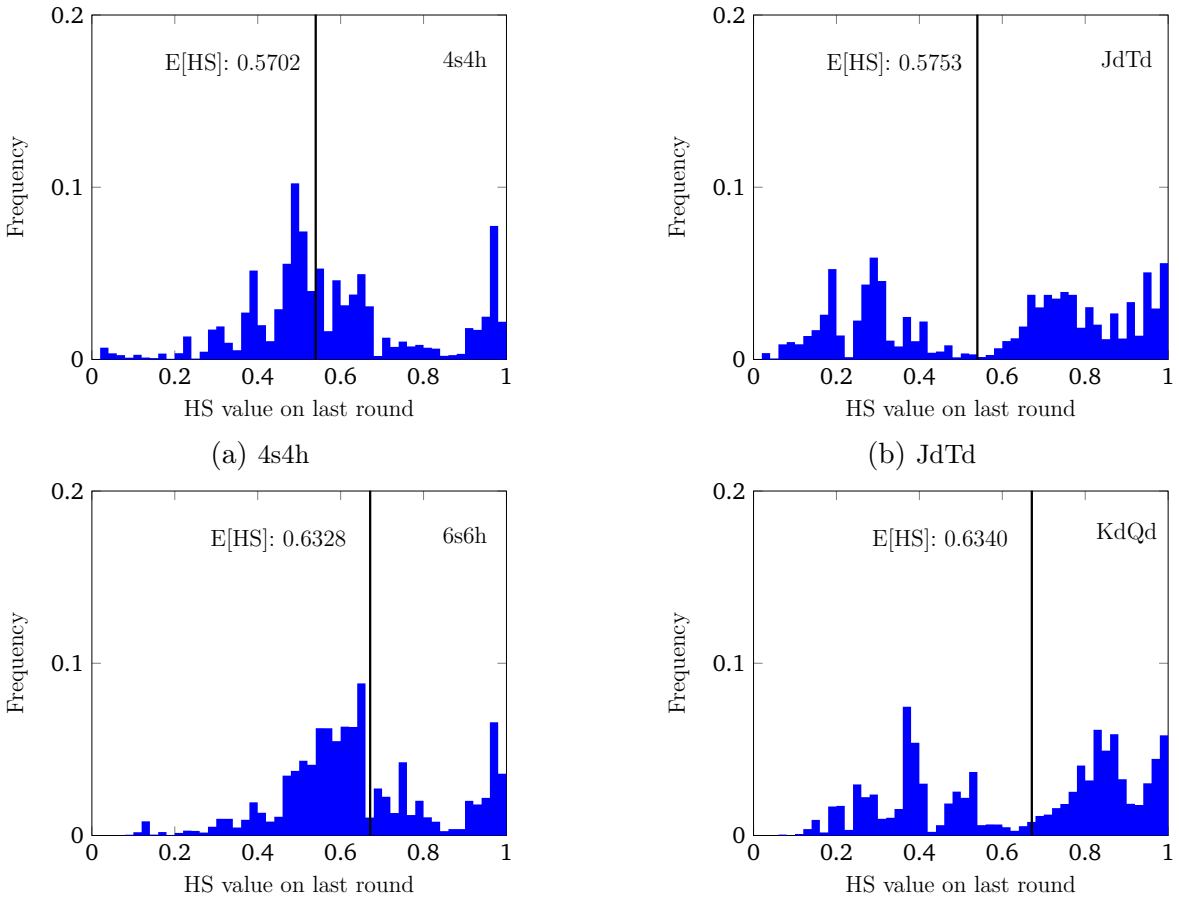
3.3.4 Nested Card Abstractions

A **nested card abstraction** is an abstraction in which hands are first grouped into buckets according to some metric and those buckets are later subdivided. One example of such a abstraction was the nested $E[HS^2]$ / $E[HS]$ abstraction which was used by the bot Hyperborean in several Annual Computer Poker Competitions [M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013]. In this case the hands were first divided by their $E[HS^2]$ value and later subdivided by their $E[HS]$ value.
A different kind of nested bucketing involves **public card bucketing**. With the approaches described so far, the structure of the public cards was ignored, the metrics focused only on the strength of the private hands. Public card bucketing first divides boards into several groups, allowing the CFR algorithm to store different strategies for each type of board. The buckets are later usually subdivided according to some metric which takes private card information into account, such as $E[HS]$, $E[HS^2]$ or a potential aware metric [M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013], which will be described in the next section.
There are several ways to group public boards, one such technique is the **distribution history aware** metric as described in [M. Moravcik, 2014], a different way is to compute features describing the structure of the board and to cluster boards according to those features. Two such features, which will later be used to create training data for deep counterfactual value networks are the draw value and the high card value, they will be described in detail in chapter 6.

3.3.5 Potential Aware Card Abstraction

While metrics such as the $E[HS]$ value are a good estimate of a hand's general strength, they do not take future potential on different boards into account. The $E[HS^2]$ metric tries to measure a hand's future potential more accurately by squaring the HS values of a hand on each river, and in doing so rewarding hands which can develop into a very strong hand and punishing hands which will be mediocre in most situations, but it was only a minor evolution of the $E[HS]$ metric.

The **potential aware card abstraction**[M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013] tries to estimate a hand's potential after future cards more accurately, it does that by first creating a probability distribution of future HS values for each hand and then clustering hands, using the **k-Means** [T. Kanungoand D. Mount, 2002] algorithm and the **earth mover's distance** [M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013].
In order to obtain the histograms each probability distribution is discretized, with values between 0 and 1, corresponding to the probability of winning against a uniform distribution. All possible future cards are then rolled out, starting from the betting round on which we need to create a bucketing. Once the histograms are created, hands get clustered into a predefined number of buckets, using the earth mover's distance. Since the earth mover's distance measures not only the difference in probability, but also the work that is required in order to transform one histogram into the other, it is better suited for the task of clustering HS histograms, as experiments have also shown [M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013].
The following example shows the differences between expectation based and potential aware card abstractions. HS histograms were created for 4 hands, starting from the preflop round and rolling out all possible cards. The hands are 4♠4♡, J♢T♢, 6♠6♡ and K♢Q♢. The following diagrams show the HS distributions and the $E[HS]$ value of each hand.



(a) 4s4h (b) JdTd

(c) 6s6h (d) KdQd

Figure 4: HS histograms for 4 hands, starting from the preflop round

4♠4♡ and 6♠6♡ both have most of their probability mass near their $E[HS]$ values, while J♢T♢ and K♢Q♢ have much less probability mass near their $E[HS]$ values.
Intuitively this means, that 4♠4♡ and 6♠6♡ will be mediocre on most river boards, as they are only small

one pair hands. J♢T♢ and K♢Q♢ have a higher chance of making a very strong hand on certain rivers, specifically if they improve to a high pair, a flush or a straight, but on many other boards they will make very weak hands, not even improving to a one pair hand.
The following tables show the $E[HS]$ and earth mover's distances of the 4 hands.

Table 2: $E[HS]$ and Earth Mover's distances

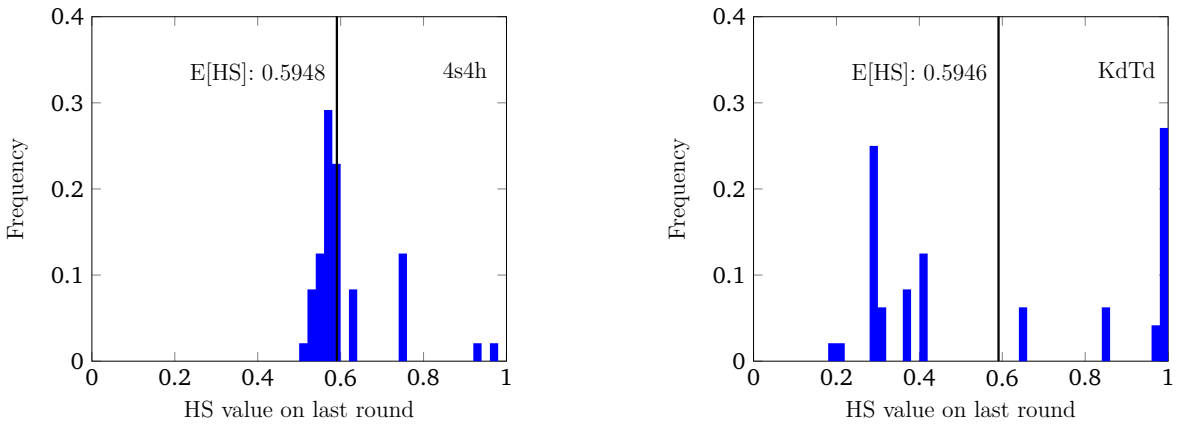
Table 3: Earth Mover's

	4s4h	**6s6h**	**JdTd**	**KdQd**
4s4h	0	3.1	6.2	5.1
6s6h	3.1	0	6.5	5.3
JdTd	6.2	6.5	0	3.1
KdQd	5.1	5.3	3.1	0

Table 4: $E[HS]$

	4s4h	**6s6h**	**JdTd**	**KdQd**
4s4h	0	0.063	0.005	0.064
6s6h	0.063	0	0.015	0.001
JdTd	0.005	0.015	0	0.059
KdQd	0.064	0.001	0.059	0

If we focus on 4♠4♡ we can see, that according to the $E[HS]$ distance the most similar hand is J♠T♠, while according to the earth mover's distance the most similar hand is 6♠6♡, therefore the earth movers distance applied to the HS histograms seems to be a better measure of the strategical characteristics of poker hands.
That intuition is supported by [M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013] in which several poker agents were created using the $E[HS]$ and earth mover's distances and a variant of CFR called **CFR-BR** [M. Johanson, N. Bard, N. Burch, and M. Bowling, 2012]. CFR-BR is able to converge to the best possible strategy, that can be expressed in a given abstraction, preventing abstraction pathologies that might occur in other CFR variants [K. Waugh, D. Schnizlein, M. Bowling, and D. Szafron, 2009] and thus reliably measuring the quality of an abstraction. It has been found, that the earth mover's distance clearly outperforms the $E[HS]$ distance [M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013].
This thesis focuses on deep counterfactual value networks on the turn round, therefore unlike in the previous example, the histograms had to be created starting from the turn.
The following example shows turn histograms for 4♠4♡ and K♢T♢ on a Q♢J♢2♡3♠ turn. Both hands have very similar $E[HS]$ values, but very different HS histograms.



(a) 4s4h (b) KdTd

Figure 5: HS histograms for 2 hands, starting from a QdJd2h3s turn board

3.4 Action Abstraction

Another way of creating a smaller, abstract version of the game of NL Hold'em, besides card abstraction, is **action abstraction**. An action abstraction only considers a subset of legal actions, thus creating a sparser version of the game tree.
There has been research on the topic of action abstractions, including [John Hawkin, Robert C. Holte, and Duane Szafron, 20012] which attempts to algorithmically find an optimal action abstraction for a given game, however most publications and most strong computer poker agents at this point use action abstractions manually designed by human experts. Most abstractions always consider the fold and call action, whenever they are legal actions in a given state and focus only on reducing the number of possible bets and raises. The bet sizes included in an action abstraction are usually expressed as a percentage of the current pot size. One such abstraction is the **fcpa** action abstraction, allowing the fold action, the call action, a raise of the size of the current pot and the all-in action. It is a popular action abstraction due to it's simplicity and the relatively small game trees it produces, as the all-in action ends the game immediately and a bet of the size of 100% of the pot will also invest all of a player's stack after only a few actions, thus ending the game.
Strong computer poker agents will often use a much finer grained action abstraction, a bot called Slumbot [E. Jackson, 2013], which ranked second in the annual computer poker competition 2016 [ACPC], used 11 bet sizes additionally to the fold and call actions.
Small action abstractions, like the fcpa abstraction are however still relevant for testing purposes and in applications with limited computational resources. The poker bot DeepStack, which will be described in detail in section 3.6 used the fcpa action abstraction in order to solve 10 million poker situations starting from the turn round, which were used as training data for its deep counterfactual value network.
As an action abstraction contains only a subset of legal actions of the game, situations in which the agent is faced with an action, which is not a part of the action abstraction can occur during game play. Unlike card abstractions the action abstraction techniques described in this section do not create a mapping of all possible actions of the full game, to actions contained in the action abstraction, during the design or training of the poker agent. Actions are rather mapped to abstract actions during game play, in a process called **action translation**. Action translation is a challenging topic, as the abstract betting tree is usually much sparser than the betting tree of the full game, even in the action abstractions used by the strongest poker bots. The bot Slumbot [E. Jackson, 2013] used an abstraction using 11 bet sizes resulting in a tree with 5.7 billion information sets, yet even such a big, state of the art action abstraction still collapsed $2.4 \cdot 10^{38}$ states of the full game tree into the same state in the abstraction [M. Johanson, 2013].
A second problem stems from the fact, that action translation errors can accumulate over the course of multiple actions, resulting in the so called **off-tree problem** [S. Ganzfried and T. Sandholm, 2015], in which the poker agent will believe to be in a vastly different state, than he actually is. This was one of the problems leading to more research in the area of action translation and techniques like endgame solving.
Similar to card abstractions, action translation can be categorized by several properties. The next subsection will talk about 2 types of action translations and give examples of concrete, popular action translations.

3.5 Action Translation

The 2 major groups of action translations are **hard translations** and **soft translations** [D.P. Schnizlein, 2009], with hard translations providing a deterministic mapping from states of the real game to states in the abstract game and soft translations providing a stochastic mapping.

Hard Translations
Formally a hard translation function is a function, which maps a betting history of the real game to a betting history of the abstract game [D.P. Schnizlein, 2009]. It can be defined using an **in-step** function, which maps a single action in the real game to an action in the abstract game. The action translation can then be implemented recursively by mapping each action in the betting history of the real game to an

action in the abstract game, in the order in which they occur, starting from the root of the game up to the current state.
The in-step function creates a mapping using a similarity metric between actions, mapping a real action to the abstract action with the highest similarity. Some popular metrics will be presented later in this section.

Soft Translations
Formally a soft translation function is a function, which maps a betting history of the real game to a set of weighted histories of the abstract game, in practice the weight represents the probability of an agent choosing a particular history [D.P. Schnizlein, 2009]. The in-state function of a soft translation maps a real action to a set of weighted action in the abstract game, the weights again representing probabilities.
A soft translation can then be defined recursively using the in-step function, similar to a hard translation.

Similarity Metrics
We will now look at 3 popular action translations, the deterministic arithmetic and the deterministic geometric action translations being hard translations and the stochastic arithmetic action translation being a soft translation.
Let a be an action in the real game and let a' and a'' be actions in the abstracted game. Let b, b' and b'' be the bet sizes corresponding to a, a' and a''. We assume, that $b' < b''$, we choose a' and a'' such, that they are neighboring actions of a, meaning $b' < b < b''$ and there exists no other abstract action with a size between b' and b''.

- The **Deterministic Arithmetic** Action Translation maps a to the smaller abstract action a' if $b < \frac{(b'+b'')}{2}$, otherwise it maps a the to bigger abstract action a''.
- The **Stochastic Arithmetic** Action Translation maps a to the smaller abstract action a' with a probability of $f_{b',b''}(b) = \frac{b''-b}{b''-b'}$, otherwise it maps a to the bigger abstract action a''.
- The **Deterministic Geometric** Action Translation maps a to the smaller abstract action a' if $\frac{b}{b'} > \frac{b}{b''}$, otherwise it maps a to the bigger abstract action a''.

Weaknesses of Action Translations
Action translations suffer from weaknesses, which can be exploited by a worst case adversary. In order to highlight potential problems, we will present an example.
Consider an agent using the deterministic arithmetic action translation and abstract actions a' and a'', with b' and b'' being the associated bet sizes for a' and a''. Let b' be equal to 10 chips and b'' be equal to 20 chips. According to the deterministic arithmetic action translation, an opponent action a in the real game, with bet size b will be mapped to a' if $b \geq 10$ and $b < 15$, this means it will be treated exactly the same as if it had in fact a size of 10. If the opponent wants to bet with a strong hand **(value betting)**, he can bet up to 15 chips without changing the agent's probability of calling compared to a bet with a size of 10. This means the opponent can gain additional value by exploiting action translation.
On the other hand if an opponent is betting with a weak hand **(bluffing)**, trying to force the agent to fold, he can make a bet with a size between 15 and 20 and it will be always mapped to a bet size of 20. This means that by making a bet of 15 chips, he can bluff cheaper without making the agent less likely to fold, again exploiting action translation.

(a) The size b is always mapped to the abstract size b'' allowing cheap bluffs

(b) The size b is always mapped to the abstract size b' allowing bigger valuebets

Figure 6: Exploting Action Translation with Bluffs and Valuebets

Stochastic action translations are harder to exploit by a worst case adversary [D.P. Schnizlein, 2009], but with any action translation the agent loses any guarantees to play a Nash equilibrium strategy of the real game.
The approach used by DeepStack, which will be described in the next section does not rely on action translation. It is always aware of the state it is currently in, preventing many of the problems associated with action translation.

3.6 DeepStack

DeepStack is a strong poker AI first described in [M. Moravcik et al., 2017], which combines traditional imperfect game solving algorithms, such as CFR+ and endgame solving, with ideas from perfect information games, while remaining theoretically sound.
This section will introduce the techniques used by DeepStack, especially depth limited continual resolving and deep counterfactual value networks.

3.6.1 Depth Limited Continual Resolving

The most common way of solving imperfect information games like poker is by precomputing a solution of an abstracted game and mapping the results back to the real game. This approach can produce strong poker AIs, but it also has many weaknesses, such as high memory requirements and the off-tree problem described in [S. Ganzfried and T. Sandholm, 2015].
In order to solve some of the problems described, several **endgame solving techniques** were proposed. An agent using endgame solving usually plays according to a precomputed strategy during the first part of the game, called the **trunk**, but computes a solution for the rest of the game, called the **endgame**, during game play. This can offer several benefits, as the original solution of the endgame can be discarded, thus saving memory space, or the endgame can be solved with a finer grained abstraction than the original endgame strategy, potentially resulting in a stronger strategy near the end of the game.
Unlike perfect information games like chess however, where solutions of sub-trees of the game can be computed independently and combined into a solution of the full game, the solution of a sub-tree in an imperfect information game can be highly exploitable in the full game.
S. Ganzfried and T. Sandholm described a technique simply called **endgame solving** [S. Ganzfried and T. Sandholm, 2015] (now sometimes referred to as **unsafe endgame solving**), which was used effectively for the bot Tartanian [A. Gilpin, T. Sandholm and T.B. Sorensen, 2008], but offers no game theoretical guarantees for the exploitability in the full game. Unsafe endgame solving assumes the opponent strategy in the trunk to be fixed and computes a solution for the endgame based on this assumption, which can be highly exploitable if the opponent changes his strategy in the trunk, as we will show in the example of **odds and evens**.

Odds and evens is a simple 2 player zero sum game. Before the game each player wagers one chip and then both players simultaneously show either an odd number of fingers or an even number, the first player wins if the sum of fingers revealed by both players is odd, the second player wins if the sum is even.
Even though both players reveal their hand at the same time, we can model the game as a sequential game, with player 1 acting first and player 2 acting second, without knowing the first player's hand. Figure 7 shows the game tree of odds and evens.

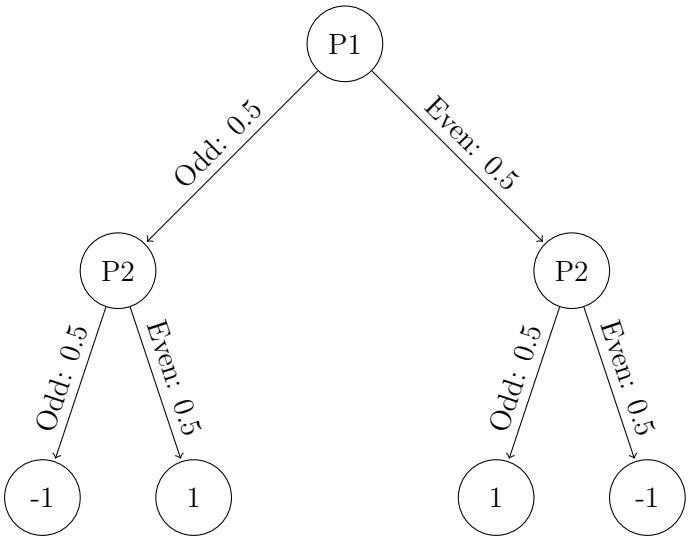


Figure 7: Game Tree for Odds and Evens with Nash Equilibrium and Payoffs for Player 1

The Nash equilibrium for this game is for both players to play odds and evens with equal probability, however the solution for the sub game starting with player 2 can be different. In the case of player 1 playing the Nash equilibrium strategy, every strategy for player 2 is a Nash equilibrium of the sub game, even a strategy like always choosing even. Every strategy will have the same expectation against a Nash equilibrium in the trunk, however many strategies are highly exploitable if player 1 is allowed to change his trunk strategy.
In order to solve this problem, [N. Burch, M. Johanson and M. Bowling, 2014] presented an endgame solving technique called **re-solving**, which is guaranteed to be no more exploitable than the original strategy, even if the opponent is allowed to change his trunk strategy.
While endgame solving techniques can improve an existing strategy, the goal of DeepStack was to never precompute a trunk strategy, it does that by using a new technique called **continual re-solving**. Continual re-solving is based on re-solving, but never keeps track of a strategy. Instead it re-solves the sub-tree starting from the current state, after every taken action. Because of that it can always compute the strategy from the exact state it is in, making action translation unnecessary and solving the off-tree problem.
While continual re-solving offers several benefits over traditional approaches, it is not in itself enough to solve a game as big as NL Hold'em HU, as it would be infeasible to compute a solution in the early stages of the game. For that reason DeepStack introduces **depth limited lookahead**. On the early rounds of the game DeepStack does not traverse the full game tree, but instead uses deep neural networks as an estimator of expected counterfactual values of each hand on future rounds for its re-solving step, resulting in the technique called **depth limited continual resolving**.
The neural networks are trained using training data created by solving random poker subgames on later rounds using CFR+, which will be described in the next section.

3.6.2 Deep Counterfactual Value Networks

In order to make Depth Limited Continual Resolving possible, DeepStack uses a deep neural network to predict the player's counterfactual values on future betting rounds. This section will give an overview of the neural networks used by DeepStack.

Training Data
Training data for the turn network was created by solving 10 million random poker situations, starting from the turn. For every situation a random public board, random private card distributions for both players and a random pot size were sampled. The pot size was sampled based on a probability distribution, which was acquired by observing NL Hold'em HU matches between Nash equilibrium based poker bots, the random private card distributions, which are represented as a vector of probabilities of a player holding a particular card combination, were created using a recursive algorithm shown in [M. Moravcik et al., 2017].
The games were solved using 1000 iterations of CFR+, which returns counterfactual values for each player and each private hand as its result. The algorithm used the fcpa action abstraction but no card abstraction.

Input
The counterfactual values of a state depend on several factors, including the pot size, the public board cards and the probability of each player holding certain private cards i.e. their distributions.
DeepStack uses the pot size at the start of the turn as one of its input feature, it is expressed as a fraction of the players' stack size.
The second feature is a representation of the players' distributions and the public cards. Before the training of the neural network starts, a potential aware card abstraction with 1000 buckets is created, as described in section 3.4. For each training example the probabilities of holding certain private hands are then mapped to probabilities of holding a certain bucket, by accumulating the probabilities of every private hand in said bucket. Because the distribution aware bucketing is created by using both, the public and the private cards and it is therefore able to differentiate between public board structures to a certain extent, no separate representation of the public cards is used.
After the training of the model is completed, the counterfactual values for each bucket in a distribution can be mapped back to counterfactual values of actual hands by creating a reverse mapping of the used card abstraction.

Architecture
DeepStack uses a standard feed forward neural network, with 1000 inputs for each player as a representation of his distribution and one additional input for the pot size, for a total of 2001 inputs.
The network has 7 fully connected hidden layers with 500 nodes each and uses parametric rectified linear units for the output.
Because NL Hold'em HU is a zero sum game, we know that the counterfactual values should sum to zero, however the values predicted by the network might not. For that reason the network is embedded into an outer network. The outer network first computes the weighted sum of the predicted counterfactual values and then subtracts half of that sum from each predicted counterfactual value, forcing the zero sum property [M. Moravcik et al., 2017].
The following image from the original DeepStack paper illustrates the architecture of DeepStack's deep counterfactual value networks.

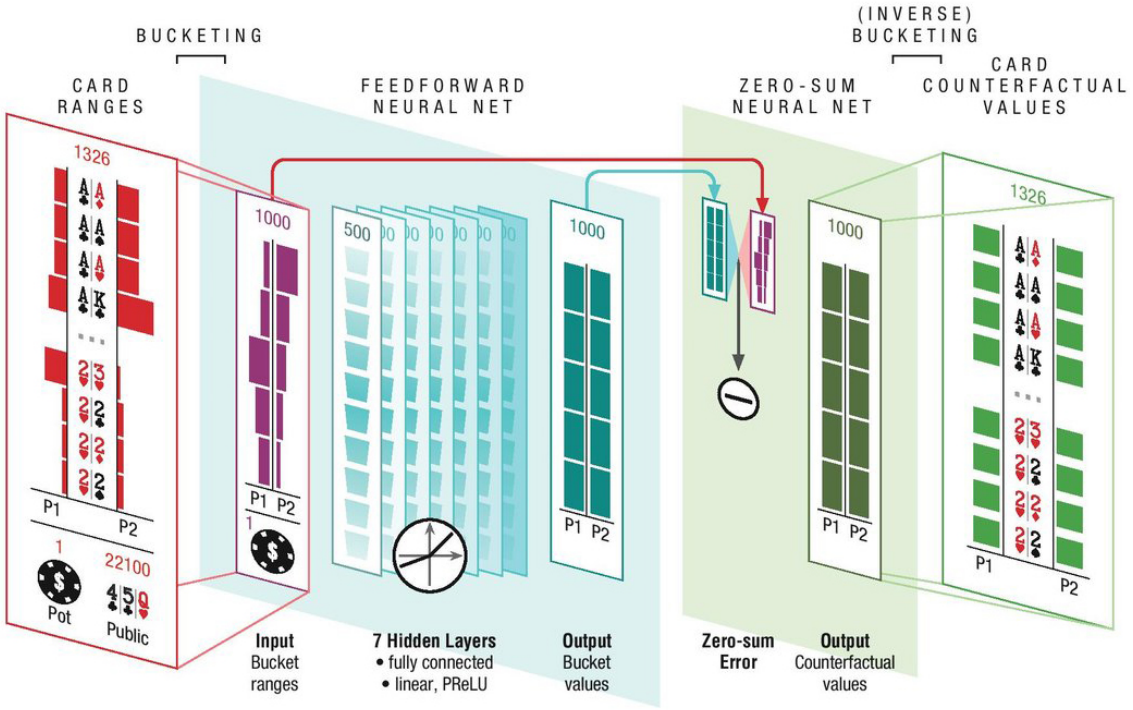


Figure 8: DeepStack's deep counterfactual value networks [M. Moravcik et al., 2017]

3.6.3 Analysis

DeepStack was able to solve many issues associated with earlier game solving algorithms. The program is always aware of its own cards during the re-solving step, thus it has solved the problem of **explicit card abstraction**. DeepStack is also always aware of the exact state it is currently in, which has solved the problems associated with **action translation** such as the off-tree problem, **explicit action abstraction** on the other hand is still used for DeepStack's sparse lookahead trees, voiding any guarantees of the algorithm converging to a Nash equilibrium of the unabstracted game [M. Moravcik et al., 2017].
Many of those improvements are only possible because of the use of depth limited lookahead using deep counterfactual value networks. It has been proven, that as the predictions of the deep counterfactual value networks come closer to the true value of a state, depth limited re-solving approaches a true Nash equilibrium of the game [M. Moravcik et al., 2017].
Deep counterfactual value networks introduce their own potential problems though, as wrong predictions of values of future states could potentially result in a highly exploitable strategy. One of the potential reasons for incorrect predictions might be the encoding of the player distributions as well as the counterfactual value outputs. The distributions and the outputs are encoded using a potential aware card abstraction, potentially leading to similar problems as traditional card abstraction techniques, which is something we will call **implicit card abstraction**, which will be analyzed in depth in chapter 5.
Another reason might be the fact, that training examples were generated using the small fcpa action abstraction, something that we will call **implicit action abstraction**, which will also be discussed in chapter 5.

4 Turn Solver

In order to create the training examples for the deep counterfactual value networks, an implementation of the CFR+ algorithm for turn subgames had to be developed.
This chapter will first provide implementation details for the algorithm, focusing on techniques necessary to improve the runtime of the algorithm.
Later the convergence of exploitability as well as counterfactual values will be analyzed, with the goal of finding the optimal number of training iterations for the CFR+ solver.

4.1 Implementation

DeepStack is the first algorithm for solving imperfect information games which combines traditional solving algorithms with limited depth lookahead, while being theoretically sound. However creating the AI was not only an advancement in game solving theory, but also a technological challenge. In order to train the turn network for depth limited lookahead 10 million random poker situations were solved with 6,144 CPU cores of the Calcul Quebec MP2 research cluster, using over 175 core years of computation time [M. Moravcik et al., 2017]. It is therefore essential to create a fast implementation of the CFR+ algorithm. This section will discuss techniques for speeding up the computation and analyze the runtime improvements of each technique.

Efficient Terminal Node Evaluation
At the leafs of the game tree, which correspond to showdowns and folds in poker, the utility for each private hand has to be computed for both players. This utility will then be backpropagated towards the root of the game, in order to compute the regrets of each action for every information set of the players.
As an example consider a showdown with the hand K♠K♡: the hand will win against some opponent hands, will tie against some opponent hands and against others it will lose. The probability of the opponent holding a certain hand in this situation is determined by his current strategy, which will generally change after each iteration of CFR+. This means, that, as strategies are constantly changing, the utilities of hands at a certain terminal node will also change, which means they can not be simply precomputed and stored in a lookup table.
In every tree, the terminal nodes will make up a substantial part of the tree's nodes, it is therefore important to compute the utilities efficiently.
Let's go back to the previous example: we computed the utility of K♠K♡ by considering the opponent distribution and calculating how often the hand will win, tie or lose. As we have to compare the hand to all opponent hands, this is an $O(n)$ operation, with n being the number of all possible hands, which is 1326. We do however not only have to compute the utility of one hand of the player, but all his possible hands, which results in an $O(n^2)$ overall complexity, when using a naive algorithm.
Efficient terminal node evaluation is a technique described in [K. Waugh, M. Zinkevich, M. Bowling and M. Johanson, 2011], which reduces the runtime complexity of terminal node evaluation to $O(n)$ compared to the naive $O(n^2)$ algorithm. We will describe the approach first presented in [K. Waugh, M. Zinkevich, M. Bowling and M. Johanson, 2011] and later also provide pseudo code, which was not present in the original work.
The naive approach considers each pair of player and opponent hands, resulting in an $O(n^2)$ algorithm, however in poker the hands at a showdown can be sorted by rank. If both distributions are sorted, for each player hand there will be regions, where the opponent's hands are weaker than the player hand, equally strong and then stronger than the player's hand. Borders between these regions can be marked in order to know the probability of a win, a loss, or a tie.
This method would already work, if distributions were independent, however the hand a player is holding influences the opponent distribution. For that reason a vector of probabilities is used, which stores the probability of each of the possible 52 cards overlapping with the opponent's distribution. This still leads to a $O(n)$ evaluation.

In order to use efficient terminal node evaluation, both distributions must be sorted by rank, which is an $O(n \log(n))$ operation (MergeSort, QuickSort), however this operation usually only has to be performed at the start of each river, not for every terminal node. In the smaller turn games, the sorted distributions for each river can be precomputed only once at the start of the training and stored in a look-up table, which improves the terminal node evaluation compared to the much bigger full game.
During the development of the CFR+ algorithm, the results of the naive algorithm were used to validate the $O(n)$ algorithm.

FastShowdown(*pHoles, oHoles, op, payoff)*;
Input:
Distribution of trained player, sorted by rank *pHoles[1326]*;
Opponent distribution, sorted by rank *oHoles[1326]*;
Opponent reach probabilities *op[1326]* ;
Payoff in this node *payoff* ;
Data:
Vector of card removed win probabilities *winCr[52]*;
Vector of card removed lose probabilities *loseCr[52]*;
Sum of win probabilities *winSum*;
Sum of lose probabilities *loseSum*;
Vector of utilities *u[1326]*
winSum = 0;
j = 0; `// opponent hand index`
`// iterate over player hands`
for i = 0 **to** 1326; **do**
  `// this loop will compute the win region of hand i, because hands are sorted by rank, hand i + 1 will have a win region of equal or greater size`
  while oHoles[j].rank < pHoles[i].rank **do**
    winSum += op[j]; `// adjust winning region as long as player hand is higher ranked`
    winCr[oHoles[i].card0] += op[i]; `// adjust blocked win probability of cards`
    winCr[oHoles[i].card1] += op[i];
    j++;
  end
  `// set utility of hand i, considering the win region and card removed win probabilities`
  u[i] = (winSum - winCr[pHoles[i].card0] - winCr[pHoles[i].card1]) * payoff;
end
`// do the same as for the win region, but start from the strongest hand and subtract utility`
loseSum = 0;
j = op.length - 1;
for i = op.length - 1 **to** 0; **do**
  while oHoles[j].rank > pHoles[i].rank **do**
    loseSum += op[j];
    loseCr[oHoles[i].card0] += op[i];
    loseCr[oHoles[i].card1] += op[i];
    j–;
  end
  u[i] -= (loseSum - loseCr[pHoles[i].card0] - losecr[pHoles[i].card1]) * payoff;
end
return u

Algorithm 1: Fast Showdown Evaluation in $O(n)$

All-In Evaluation

The last section provided an explanation and pseudo code of efficient terminal node evaluation. When the game ends, an $O(n)$ algorithm can be used to compute the utilities for each player. However unlike the case of FL Hold'em HU, which was the subject of [K. Waugh, M. Zinkevich, M. Bowling and M. Johanson, 2011], in NL Hold'em players can go all in on an earlier round than the river.

In order to compute the utility of such a situation, the straightforward way would be to roll out all possible river cards and average the utilities for each hand. We can however take advantage of the smaller size of the turn game we are trying to solve and create look-up tables for the win probabilities of each hand against every other hand. Since we start from the turn and on the river we can use efficient terminal node evaluation, we will only need one look-up table per training example. The look-up table can be created prior to training in only a few seconds, which is negligible compared to the usual runtime of the training algorithm. Pseudo code for the creation of the look-up table is provided in the appendices.

TurnAllIn(*pHoles, oHoles, op, turnEquities, payoff*);

Input:

Distribution of trained player *pHoles[1326]*;
Opponent distribution *oHoles[1326]*;
Opponent reach probabilities *op[1326]* ;
Matrix of hand vs. hand equities *turnEquities[1326][1326]*;
Payoff in this node *payoff* ;

Data:

Sum of win and lose probabilities *winSum*;
Vector of utilities *u[1326]*

```
for i = 0 to 1326; do
    winSum = 0;
    for j = 0 to 1326; do
        if !pHoles[i].overlaps(oHoles[j]) then
            winSum += op[j] * (2 * turnEquities[i][j] - 1);
        end
    end
    u[i] = winSum * payoff;
end

return u
```

Algorithm 2: Turn All-In Evaluation

Static Memory

Every implementation of a CFR algorithm must store the regret values and, if used, the average strategy for every information set, as they are the result of the algorithm and must be updated over several iterations. The memory for the regrets and the average strategy is allocated once at the start of the training and no new memory for the regrets or average strategy is allocated afterwards.

Since a full poker game has a big game tree, even when using a state of the art abstraction, memory is very limited and allocating memory for auxiliary data structures like look-up tables is very costly.

The last 2 sections provided examples in which we can take advantage of the smaller size of the turn subgame and trade memory for speed, namely by creating a look-up table for hand vs. hand win probabilities and by storing the sorted player distributions for every possible river.

There are however even more ways in which the allocation of memory during the training can be avoided. Examples of this include the reach probabilities, which are updated for every player decision and passed to a child node, as well the utilities for each player, which are backpropagated from the terminal nodes back to the root of the game. In the small turn game the memory for those vectors can be allocated for each

node of the betting tree prior to training and never has to be deleted or reallocated afterwards. During training only a reference to the vector has to be passed.
Another possible optimization is related to the pot size sampling and the creation and deletion of game trees. Different starting pot sizes result in slightly different game trees, as for example a bigger starting pot will result in fewer possible actions according to the fcpa action abstraction. Because the creation and deletion of game trees is costly, the implemented algorithm only samples a pot size at the start of the program, after that the pot size stays constant and only the distributions and board cards get re-sampled for each new training example. In this way a game tree has only to be created once and it does not to be deleted either. This method could potentially lead to the problem of too few pot sizes being sampled, but because many processes were running in parallel, each sampling a different pot size and they were restarted frequently, all possible pot sizes were sampled.
The CFR+ implementation used for this thesis never allocates any memory, except for primitive values, during training, which lead to a substantial improvement in runtime. The effects of this technique as well as fast terminal node evaluation and turn all-in evaluation will be shown in the next section.

Runtime Analysis
In order to measure the effect of each technique described in this section, runtime tests were conducted after every optimization step. The values for the baseline implementation, which contained no optimizations and for the algorithm with only memory optimizations were averaged over 10 thousand iterations of the CFR+ algorithm.
The results after the fast terminal node evaluation and the turn all-in evaluation were implemented, were averaged over 100 thousand iterations.

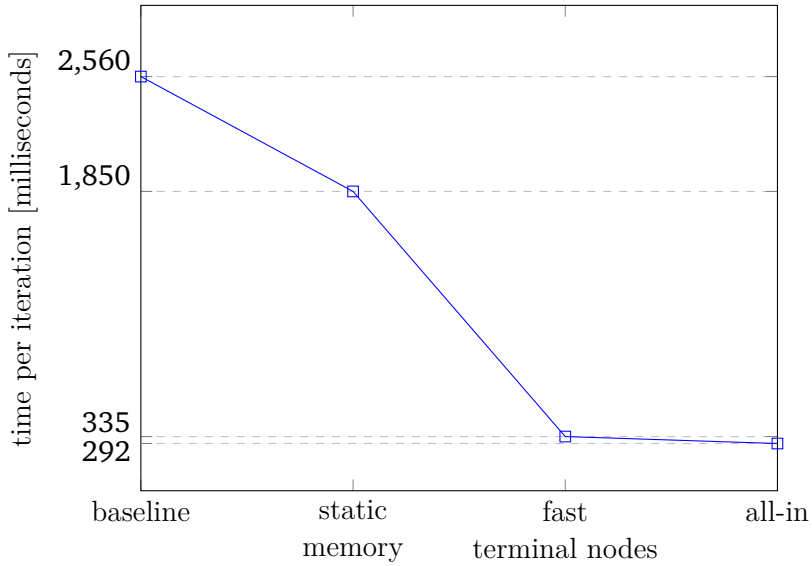


Figure 9: Runtime Optimizations

Every optimization was able to improve the runtime of the algorithm, with the memory optimization leading to an improvement of around 28%, the fast terminal node evaluation improving the runtime again by around 82% and the all-in evaluation reducing the runtime by another 13%. The improvement of all the optimizations combined was 89% compared to the baseline implementation.

Table 5: Runtime Improvement

	Baseline	**Static Memory**	**Terminal Nodes**	**All-In**
Runtime	2560	1850	335	292
Relative Improvement	0%	28%	82%	12%
Total Relative Improvement	0%	28%	87%	89%

4.2 Evaluation

The training examples for DeepStack were created by approximately solving random turn situations using 1000 iterations of the CFR+ algorithm. The result of the CFR+ algorithm approaches a Nash equilibrium and generally improves with a higher number of iterations.
In practice many compromises have to be made, as computational resources are limited. A higher number of iterations might produce more accurate training data, but on the other hand the number of training examples will be lower.
This section will analyze the convergence of the CFR+ algorithm using the raw training data, specifically the exploitability and the counterfactual values produced by the algorithm. The tests were performed on a small data set in order to find the right number of iterations which was then used to create the much bigger data set for more detailed tests, which will be described in chapter 5.

Exploitability
This section will analyze the improvement of exploitability over an increasing number of iterations. For that purpose 2000 turn situations were solved, using 5000 iterations of CFR+ and measuring the exploitability, as a fraction of the pot size, after every 200 iterations.

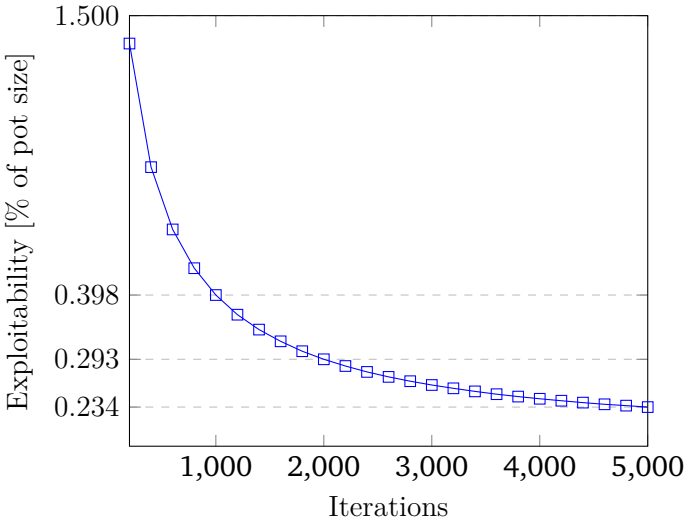


Figure 10: Convergence of Exploitability

The following table shows the exploitability after every 1000 iterations, expressed as a percentage of the pot size. It also shows the relative improvement in exploitability compared to the solution in the previous

column as well as the overall relative improvement compared to the solution with 1000 iterations.

Table 6: Exploitability Improvement

	1000 iterations	**2000 iterations**	**3000 iterations**	**4000 iterations**	**5000 iterations**
Exploitability	0.398%	0.293%	0.259%	0.243%	0.233%
Relative Improvement	0%	27%	12%	7%	4.1%
Total Relative Improvement	0%	27%	35%	39%	42%

The average exploitability after 1000 iterations, which is the number of iterations, that DeepStack used, was 0.398% of the pot size, the exploitability after 2000 iterations was on average 0.293%, which is a relative improvement of about 27%. This shows, that the solution is still improving at a high rate at that point.
As the number of iterations increases, the relative improvement of the exploitability decreases, the reduction of exploitability between a solution after 5000 iterations, compared to a solution after 4000 iterations is only about 4.1%.

Counterfactual Values
While the exploitability of a strategy provides a very good measure for its quality, deep counterfactual value networks try to predict counterfactual values of a strategy, rather than its exploitability. In this section we will analyze the counterfactual values directly.
For that purpose the same 2000 turn situations as in the previous section will be used. The solution which used 5000 iterations will be used as a reference, and its counterfactual values will be compared to solutions with fewer iterations.
We will use the mean squared error as the similarity measure, for every solution the mean squared error of its counterfactual value vector and the counterfactual value vector of the reference solution will be computed. The following graph shows the development of the MSE.

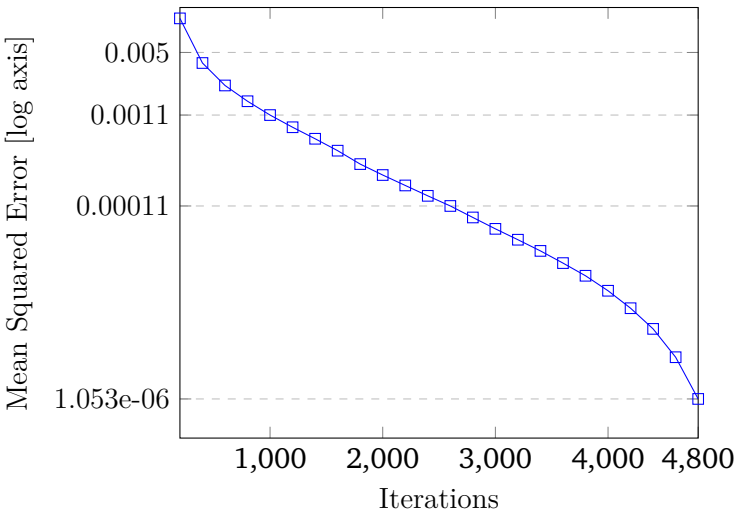


Figure 11: Convergence of MSE

The following table shows the MSE after every 1000 iterations, compared to the reference solution with 5000 iterations, as well as the relative improvement over the prior solution and the total improvement over the solution with 1000 iterations.

Table 7: MSE Improvement

	1000 iterations	**2000 iterations**	**3000 iterations**	**4000 iterations**
MSE	0.0011	0.00025	6.72e-05	1.48e-05
Relative Improvement	0%	77%	73%	77%
Total Relative Improvement	0%	77%	93%	98%

We can see, that the MSE monotonously approaches zero as the number of iterations increases. Even after 3000 iterations, the relative decrease of the MSE is still high with 73%. The improvement over the strategy with 1000 iterations slows down however, with the strategy with 4000 iterations only gaining about 5% compared to the solution with 3000 iterations.

4.3 Summary

The purpose of this section was to determine the number of iterations, that the training algorithm should use for the final data set.
At around 1000 iterations the exploitability of the strategy is still decreasing rapidly, even after 5000 iterations, which were the limit for this test, the strategy has not yet converged very well, although improvement slows down.
The MSE of a solution compared with the reference solution after 5000 iterations also improves at a high rate, with a relative improvement of over 70% after each 1000 iterations.
Both metrics indicate, that a number of iterations higher than 1000 might be optimal. The main neural networks which were created during this thesis and which will be discussed in more detail in chapter 5, used training examples which were trained for 2000 iterations.
Using 2000 iterations has a few advantages over using 1000 iterations, the first being that the exploitability is 27% lower on average, the second being that it allows us to analyze the effect of the number of iterations compared to DeepStack, chapter 5 will analyze how the higher number of iterations changes the neural network model as well as the trade-off between the number of iterations and the data set size.

5 Deep Value Networks

This chapter will focus on our version of deep counterfactual value networks. First an overview of the implementation of our baseline network will be given, which had the goal of closely mimicking DeepStack's own implementation.
Later several aspects of the neural network will be analyzed, some of which we will try to improve upon in chapter 6.

5.1 Baseline Implementation

The goal for this implementation of a deep counterfactual value network was to follow DeepStack's own implementation as close as possible in order to analyze the techniques used by the poker bot. We will now describe the implementation, highlighting similarities and differences compared to DeepStack.

Framework
The original implementation of DeepStack used the **Torch7** [Torch Library] library for the programming language LUA to build and train its neural networks [M. Moravcik et al., 2017]. Due to unfamiliarity with the LUA programming language, the popular library **TensorFlow** [TensorFlow Library] was used for this thesis instead.
TensorFlow is an open source library for numerical computation and machine learning, which was officially released in 2015 and is strongly Supported by Google. It offers implementations for several programming languages, including Java, Python, C++ and Go, for this thesis the most popular Python implementation was chosen.
The library relies on a highly optimized C++ and CUDA back-end in order to achieve maximum performance and supports the training of neural networks on CPUs as well as GPUs.

Distribution Encoding
Like the original implementation of DeepStack the networks created during the work on this thesis used the potential aware card abstraction [M. Johanson, N. Burch, R. Valenzano and M. Bowling, 2013] described in chapter 3 to encode the players' distributions. The number of buckets was 1000, which is equal to the number of buckets used by DeepStack. The bucketing was created using an imperfect recall abstraction, which means that the order of the public board cards was ignored.
In order to create the buckets, first river HS histograms with 50 bins were created for all possible canonical hands. A multi-threaded implementation of the k-Means [T. Kanungoand D. Mount, 2002] algorithm was then used to cluster the hands into 1000 buckets according to the earth mover's distance of their HS histograms.
The k-Means implementation used the k-Means++ [D. Arthur and S. Vassilvitskii, 2007] initialization, which spreads out the initial cluster centers with the goal of achieving better results as well as the triangle inequality [C. Elkan, 2003] which can improve the runtime substantially.
The clustering was performed for 1000 iterations, which required several hours of runtime, despite the multithreaded implementation and the triangle inequality, however the clustering has only to be performed once prior to the training of a network and can be reused for other networks.
As a final step the distributions had to be encoded, using the bucketing. Every training example created by the CFR+ turn solver was created with random private card distributions for each player, those distributions were represented as a vector of probabilities, representing the probability of holding each possible hole card combination. The probabilities of holding certain hole cards were then mapped to probabilities of holding a certain bucket, by summing the probabilities of all hole cards in that bucket.

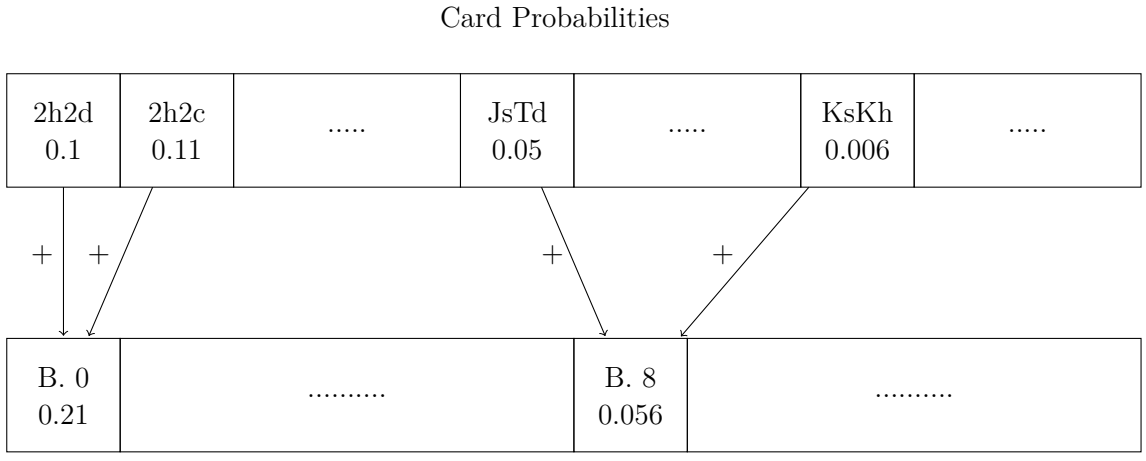


Figure 12: Encode Distribution By Summing Card Probabilities

Similar to the input, the output also had to be encoded. The CFR+ solver returns the counterfactual values of each possible hole card combination, those counterfactual values of hole cards had to be mapped to counterfactual values of buckets. This can be achieved by simply averaging the counterfactual values of each hole card combination in that bucket. The effects of that encoding will be analyzed in depth in section 5.2.

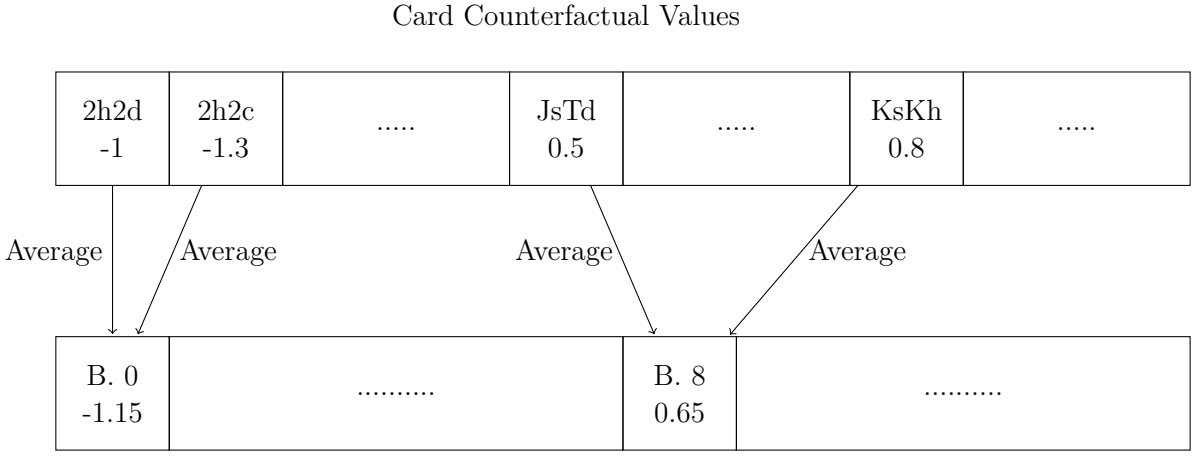


Figure 13: Encode Output By Averaging Card Counterfactual Values

Turn Sampling

In order to create the training examples for the network, random turn situations had to be sampled. The random turn situations can be characterized by 3 variables: the pot size, the turn board and the players' private card distributions.

In our implementation the board cards were sampled according to a simple uniform distribution, the sampling of private card distributions followed the same pseudo random algorithm, that DeepStack used [M. Moravcik et al., 2017], there was however a difference in the sampling of pot sizes.

DeepStack samples pot sizes according to a distribution obtained by observing the play of earlier poker

bots, while for this thesis a distribution based on the play of human poker players was chosen. The reasons for that and the implications will be discussed in detail in section 5.7.

Training Data
Unlike DeepStack, which used about 10 million training examples to train its counterfactual value network, our network used only about 300 thousand training examples. The main reason for that were computational limitations, a second reason was, that our training examples were trained using 2000 iterations, unlike DeepStack which used examples trained for only 1000 iterations.
The original data was used to create 2 data sets, the **training set** consisting of 80% of the examples and the **test set** consisting of 20% of the examples.
As described, the counterfactual values of each private hand were encoded using a potential aware card abstraction and all training happened on that encoded dataset. When measuring the precision of the model, we have therefore 2 possible perspectives. The first is to look at the prediction error with both inputs and outputs encoded with the card abstraction, the second way is to map the predictions of buckets back to predicted counterfactual values of private hands. The predicted counterfactual values of private hands can then be compared to the unabstracted counterfactual values of the test examples.
When measuring the error using encoded inputs and outputs, we will refer to the test set as **abstract test set**, when we are measuring the prediction error for unabstracted private hands, we will call the dataset the **unabstracted test set**.
We will use the same logic for the training set. While the unabstracted training set is never actually used for training, as all training happens only on the encoded inputs and outputs, it can still be interesting to measure the prediction error on the unabstracted training set and compare it to the unabstracted test set.

Architecture and Training
The architecture of the implemented neural network is identical to that of DeepStack, the network uses 7 fully connected hidden layers and parametric rectified linear units. Just like the original network it is embedded into an outer network which forces the zero sum property of the game [M. Moravcik et al., 2017]. The network was trained for 350 epochs using the Adam Gradient descent and the Huber Loss. The Huber loss is defined as

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{if } |y - f(x)| < \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

It is quadratic for small prediction errors and linear for prediction errors, which are bigger than the hyper parameter δ. For that reason it does not punish outliers as strongly as the MSE for example, which can result in a more robust regression. The following Figure shows a graph of the Huber Loss with $\delta = 1$, compared to the MSE.

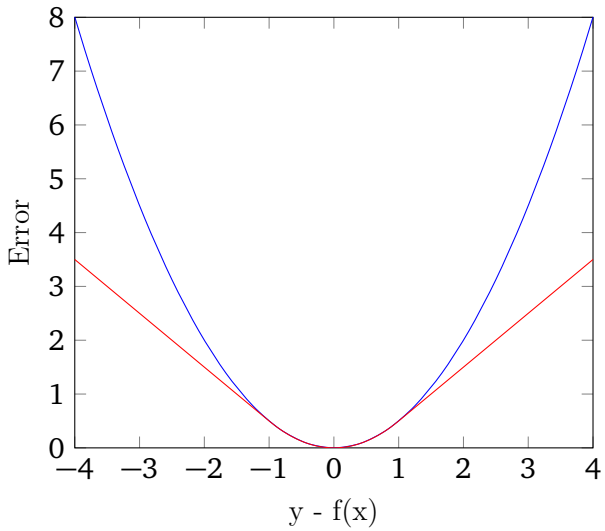


Figure 14: Huber Loss (Red) compared to MSE (Blue)

The original DeepStack implementation used a validation set in addition to a training set and a test set. It saved a neural network model after each epoch and chose the model with the lowest validation error as the final model [M. Moravcik et al., 2017]. The deep counterfactual value network implementation in this thesis did not use a validation set, the first reason for this was the smaller data set size, which lead to the decision to use more examples for the training set instead. The second reason was, that in early testing the error did not change much after a certain number of iterations, so it did not seem as if a validation set would strongly improve the performance of the network.

Hardware

The 10 million training examples for DeepStack were created on 6144 CPU cores of the Calcul Québec MP2 research cluster, using over 175 core years of computation time [M. Moravcik et al., 2017]. The 300 thousand training examples for this thesis were created using on average roughly 150 CPU cores on the Lichtenberg cluster of the Technical University of Darmstadt, using only about 40000 core hours.
The actual training of DeepStack's network was performed on a single GPU [M. Moravcik et al., 2017] although it is not specified which GPU was used, for this thesis an NVIDIA GeForce GTX 1080 Ti was used.

Results

The following table shows the results of the implemented network.

Table 8: Prediction Error

Error Abstract Training Set (Huber Loss)	0.0052
Error Unabstracted Training Set (Huber Loss)	0.0267
Error Abstract Test Set (Huber Loss)	0.0102
Error Unabstracted Test Set (Huber Loss)	0.0297

The network achieved a Huber loss of 0.0052 on the training set, 0.0102 on the abstract test set and 0.0297 on the unabstracted test set.

DeepStack achieved a Huber loss of 0.016 on its training set and 0.026 on its test set [M. Moravcik et al., 2017], although it is not clear if its test set would correspond to the abstract test set or the unabstracted test set by our definitions. The different results of the abstract test set and the unabstracted test set will be analyzed in detail in the next section.
One goal of this thesis was to compare the prediction error of our deep counterfactual value network implementation to the results achieved by DeepStack, there are however, several difficulties. As already mentioned, it is not clear, if the original DeepStack paper talked about the error on the abstracted data set, or was testing the counterfactual value predictions for actual hands. Furthermore some hyper parameters, like the parameter δ of the Huber Loss were not specified, which might change the results compared to our implementation, which used the TensorFlow default of $\delta = 1$.
An additional difference between the implementations is probably, that we were forced to use a weight matrix for the training, the reasons for which will be explained in the next section. Finally DeepStack used a much bigger data set.
Because of those reasons, a direct comparison to the results produced by the original DeepStack implementation are unfortunately not possible. The following chapters will analyze potential ways of improving the techniques used by DeepStack, we will however, evaluate all results by comparing them to our baseline implementation and not to the original DeepStack implementation.

5.2 Implicit Card Abstraction

One of the biggest problem of traditional imperfect information game solving algorithms is the need for abstraction to produce solutions for human scale problems. In the case of card abstraction problems can arise from the fact, that the agent is not aware of the actual cards that he is holding. Results using a greedy best response approximation called **local best response** [M. Moravcik et al., 2017] [M. Bowling, , 2017] suggest, that traditional poker bots are highly exploitable by an agent that has perfect information about his cards.
Therefore one of the main goals of DeepStack was to not use card abstraction. DeepStack never uses explicit card abstraction during it's continual re-solving step, but is instead aware of the exact cards he is holding. The training examples for DeepStack's deep counterfactual value networks also don't use any card abstraction, however the encoding used for the networks creates a problem that can be called **implicit card abstraction**.
There are two ways in which implicit card abstraction happens, the first one is the encoding of the input distributions. Because the player distributions get mapped to a number of buckets prior to training, the training algorithm is not aware of the exact distributions, but only of the distribution of bucket probabilities. Because this is a many to one mapping, the algorithm might not be able to distinguish different situations, thus not being able to perfectly fit the training set.
The second problem stems from the encoding of the output values. Counterfactual values of several hands are aggregated to a counterfactual value of a bucket, potentially losing precision.
While the intuition behind the encoding of the inputs and outputs is similar, the error introduced by the encoding of the outputs is easier to analyze and will therefore be the focus of this section.

Bucket Analysis
DeepStack as well as our implementation both use a card abstraction with 1000 buckets. This number seems high at first, as there are only 1326 possible private card combination and after the turn cards were dealt only 1128 of them are still possible due to overlapping cards, but we have to keep in mind, that the bucketing is an encoding of both private and public cards.
Ignoring the order of board cards there are 305377800 possible hand combinations on the turn, which, using isomorhisms, can be reduced to 13960050 isomorphic indexes [K. Waugh, 2013], which means, that on average 305377 hands or 13960 isomorphic indexes are mapped to the same bucket.
Because the card abstraction is created using both the public and private cards, not all buckets will be present on every public board, on a given public board private cards might be mapped to only a small

subset of all possible buckets.
For the following chart all possible turn board combinations were enumerated and the number of possible buckets on that board was recorded.

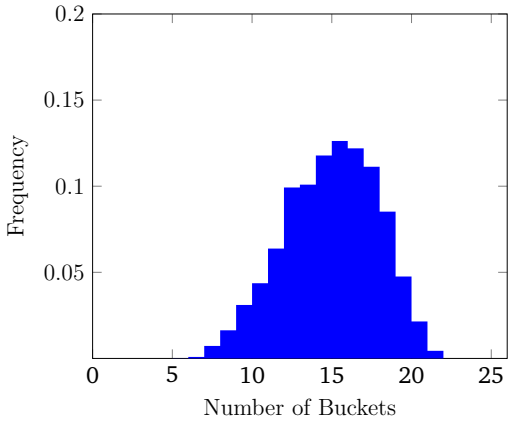


Figure 15: Distribution of Number of Buckets per Turn

The number of buckets, which are possible on each turn is on average much lower than the total of 1000 buckets, with the maximum number of buckets being only 22.

Table 9: Number of Buckets per Turn

Min	**Max**	**Mean**	**Median**
5	22	14.47	15

On each turn board, 1128 private card combinations are possible, which will be mapped to the given number of possible buckets on that board. In reality some of those private cards will be isomorphic to each other, meaning that the number of strategically different cards which are mapped to a given bucket will be slightly lower. Because of a different number of suits present on different boards, the number of private card isomorphic indexes on each board can differ. The number of private cards indexes was analyzed similar to the number of possible buckets on each board, the results are listed in the following table.

Table 10: Isomorphic Indexes per Turn

Min	**Max**	**Mean**	**Median**
144	1080	849.6	684

The results of both experiments show, that on average 849.6 strategically different hands are mapped to 14.47 buckets on that board.
Additionally to analyzing the number of buckets used on each turn, we can also analyze the reversed question of how many hands are mapped to a certain bucket and how many turns use a certain bucket. This can give us some insight into the quality of our version of the potential aware card abstraction, as buckets which are never or very rarely used do not increase the accuracy of the abstraction significantly.
The following chart shows the number of canonical hands mapped to each bucket, ordered by the size of the bucket

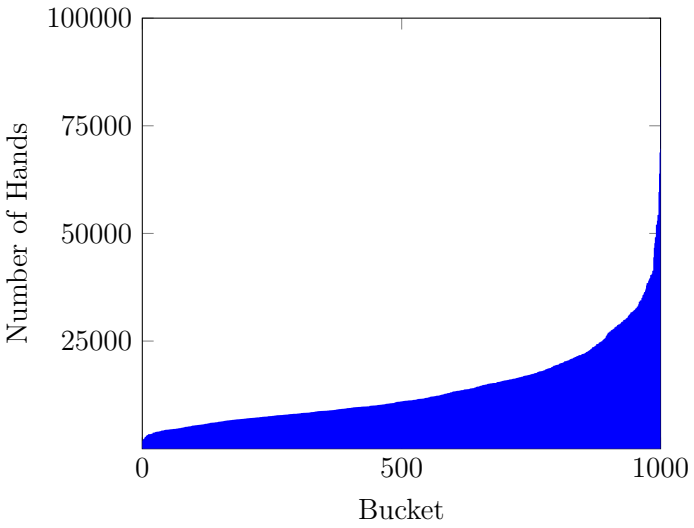


Figure 16: Number of Hands mapped to a Bucket

We can see, that the number of hands mapped to each bucket varies, which is to be expected as groups of strategically similar hands will likely have different sizes as well. There is however no bucket, which is never used and even the least used bucket is fairly close to the mean.

Table 11: Hands per Bucket

Min	**Max**	**Mean**	**Median**	**Std Dev**
8044	88702	13960	11004	11824

The following chart shows the number of turns, which use a certain bucket, the buckets are again sorted by the number of turns on which they are represented.

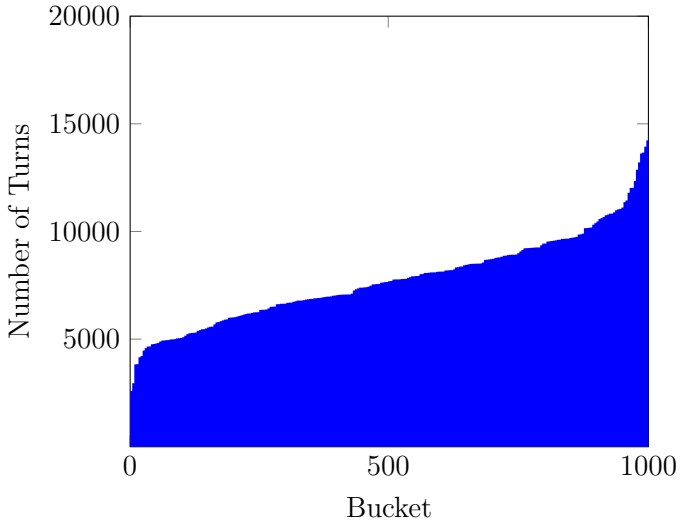


Figure 17: Number of Turns using a Bucket

It can be again observed, that while the number of turns using a bucket varies, every bucket is represented on several turns and the distribution is fairly even, with about 79% of buckets being present on between 5000 and 10000 turns.

Table 12: Turns per Bucket

Min	Max	Mean	Median	Std Dev
2598	15027	7791	7698	2102

In this section we were able to see, that the number of hands mapped to a certain bucket is high and each turn only uses a small subset of buckets. This is not due to unused or very infrequently used buckets, but seems to be a result of the small overall number of buckets as well as the properties of the potential aware card abstraction.
In the following the implications of the findings will be analyzed, first the loss of accuracy will be examined empirically, later an example will show, that the problems are not exclusive to the potential aware card abstraction.

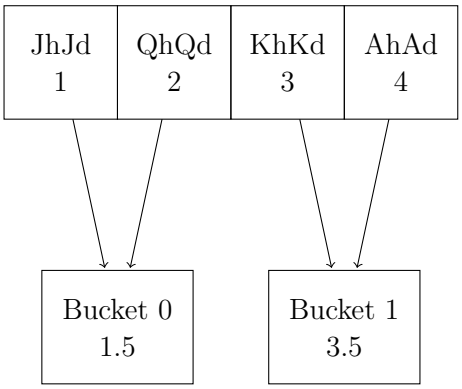
Encoding Error
For the experiment the counterfactual values of the unprocessed data were compared to the counterfactual values of the corresponding buckets in the processed training examples. All 300,000 endgame solutions of the baseline implementation were used for the test. As similarity metrics both the MSE as well as the Huber loss were computed for each training example and then averaged.

Table 13: Encoding Error

MSE	Huber Loss
0.0258	0.0544

We can see, that the encoded training data does not perfectly represent the counterfactual values of each private card combination. The encoding error of the potential aware card abstraction will be compared to other traditional abtsractions in chapter 6 and a new abstraction, created for this thesis, in chapter 7.
While the encoding error will depend on the kind of card abstraction used as well as implementation details, a small number of buckets per turn will generally lead to an encoding error greater than zero, which we will show in an example.
In the following example we try to map the counterfactual values of four hands to two buckets

Card Counterfactual Values



Bucket Counterfactual Values

Figure 18: Example Bucketing

In the example J♡J♢ and Q♡Q♢ are mapped to bucket 0 and K♡K♢ and A♡A♢ are mapped to bucket 1, the average counterfactual values are 1.5 and 3.5 respectively. The mean absolute error of the card counterfactual values compared to the counterfactual values of their buckets is 0.5.
In this example there exists no method to map the cards to buckets in such a way, that there is no encoding error. When we see the card combinations as a sorted list, we can try to find a split index in order to create two lists, representing the two buckets, with all cards before the split index belonging to bucket 0 and all other cards belonging to bucket 1, the following table shows all possible splits and their mean absolute errors

Table 14: Error of Possible Bucketings

Split Index	**0**	**1**	**2**	**3**	**4**
Mean Absolute Error	1	0.5	0.5	0.5	1

The optimal split indexes in this example are 1, meaning that J♡J♢ is in in the first bucket and the rest of the cards is in the second bucket, two, which is shown in the example and three, which means only A♡A♢ is in the second bucket, while the rest of the cards is in the first bucket. In all cases the encoding error is greater than zero however.
The example shows, that there are situations, where a lossless bucketing can not be achieved with a number of buckets lower than the number of cards.
In chapter 6 we will test other encodings based traditional card abstractions and analyze the effect of factors like the number of buckets present on each turn. We will also try to feed hand features directly into the neural network, hopefully solving some problems, which might occur as a result of a small number of buckets.

Sparse Inputs and Undefined Outputs
While we have already shown, that as a result of the aggregation of hand counterfactual values into bucket counterfactual values, an encoding error is introduced into the training data, another problem occurred while working with the potential aware encoding.
Because of the low number of buckets used for each training examples, a way of representing the inputs and outputs for buckets, which are not present on the given turn public board, has to be decided. The first way of encoding the output counterfactual values for those buckets was to set them to zero. This

representation seems intuitive, but it led to very bad neural network models and does also not make much sense upon further investigation. Logically the counterfactual value of a bucket, which is not present on a given board, is not defined. This means, that setting it to zero is a mistake, especially as there might actually be buckets present on that board, which have a counterfactual value of zero or very close to zero. Additionally to theoretical concerns, the neural networks also produced very bad predictions, predicting a counterfactual value of zero in most cases.
The solution which was used for this problem was a weight matrix, which acted as a mask, as the weights of buckets which are not present on a given board were set to zero and all other weights were set to one. As a result of using this weight matrix, the optimizer did ignore the loss of outputs, which were not defined on a board and the results improved dramatically.
The probabilities of input buckets, which are not present on a given public board, were also set to zero. Logically it makes more sense, than setting the outputs to zero, as the probability of a player holding a hand in that bucket is indeed zero. There is however still some difference between a player not holding a hand in a certain bucket, due to his strategy, or due to the fact, that said bucket is not possible on a board. Despite theoretical concerns, encoding the input probabilities of those buckets as zero did not seem to decrease the network accuracy. Other representations were tried, including setting the inputs to -1, but no improvement was recorded, so in the end buckets, which were not present on a board, were assigned a probability of zero.
As mentioned when presenting the results of section 5.1, this is one of the reasons, why it is difficult to compare our results to the results of the original DeepStack implementation.

5.2.1 Summary

In this section we have observed, that with the given encoding scheme, several hands get mapped to the same bucket. We have seen, that empirically this produces a difference in the original training data and the encoded training data. The difference was not simply a result of implementation details, but a direct result of a small number of buckets.
With the inaccuracy introduced to the training data, even a hypothetical perfect neural network would not be able to produce perfect predictions for counterfactual values of the actual hands, we can therefore conclude that the encoding scheme does introduce a form of implicit card abstraction. Chapter 6 will attempt to improve on the encoding used for DeepStack.

5.3 Implicit Action Abstraction

Unlike explicit card abstraction, DeepStack still uses explicit action abstraction for its sparse lookahead trees [M. Moravcik et al., 2017] during its re-solving step.
Similar to card abstraction however, it introduces a problem that we will call **implicit action abstraction** additionally to its explicit action abstraction. The training examples for the counterfactual value network were solved using the fcpa action abstraction, which can produce counterfactual values different from those of a real Nash equilibrium. This potential problem will be analyzed in this section.

Experimental Setup
In order to analyze the effect of action abstraction on the counterfactual values of the training examples, three new action abstractions were defined and their counterfactual value results compared to solutions computed using the fcpa abstraction. The new action abstractions were

- **Abstraction 1:** fold / call / 0.5 Pot / Pot / All-in
- **Abstraction 2:** fold / call / 0.5 Pot / 0.75 Pot / Pot / All-in
- **Abstraction 3:** fold / call / 0.5 Pot / 0.75 Pot / Pot / 2 Pot / All-in

For each action abstraction 250 identical turn situations were solved and compared to the fcpa abstraction using the MSE.

Conclusions
The following graph shows the MSE of each action abstraction, compared to the fcpa abstraction. Differently from the encoding of neural network outputs, which were encoded as a fraction of the pot, we used the counterfactual values directly for this test

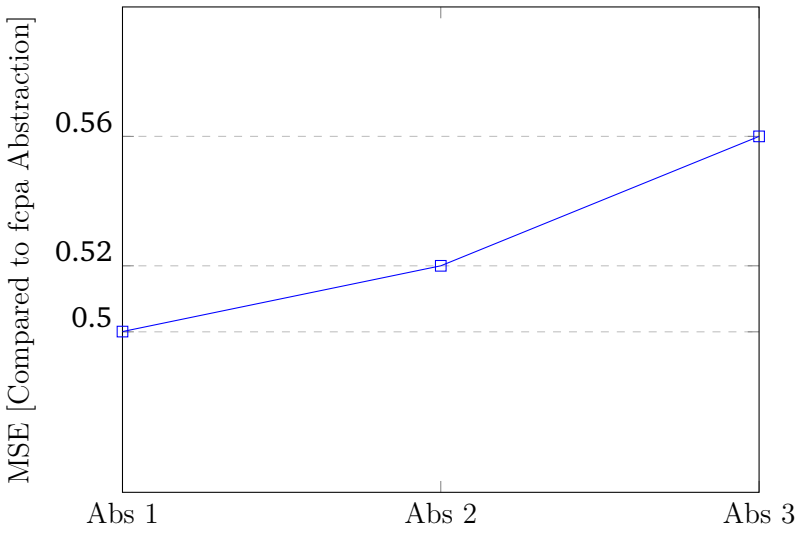


Figure 19: MSE of Action Abstractions

It can be observed that the results created with different action abstractions differ from each other, with the finest grained abstraction 3 having the highest MSE when compared to the fcpa abstraction. Even though abstraction pathologies might exist when using finer grained abstractions and no solution using action abstraction is guaranteed to converge to a Nash equilibrium of the real game, generally bigger abstractions tend to produce better results. As the abstractions get finer grained, their result differs more strongly from the fcpa abstraction, suggesting that the solution of the fcpa abstraction is far from a true Nash equilibrium.
Because of the theoretical results, saying that a abstracted solution is not guaranteed to converge to a Nash equilibrium of the real game and because of the empirical results, showing the MSE compared to finer grained abstractions, we can conclude, that the method used does introduce a form of implicit action abstraction, especially when using a small action abstraction like the fcpa abstraction.

5.4 Pot Size Sampling

The training examples for DeepStack's deep counterfactual value networks were created using random board cards, random hole card distributions and a random pot size. The pot size was sampled from a distribution acquired from observing older NL Hold'em HU programs. It is not specified which poker bot was used to create the distribution of pot sizes.
In this section we will first analyze possible distributions of pot sizes and suggesting alternatives to the approach of DeepStack and later analyze the effect of the sampling scheme on the accuracy of the network.

5.4.1 Observed Distributions

This section will analyze pot sizes of a Nash equilibrium based poker program, which was developed for an earlier thesis and was a contestant in the 2016 Annual Computer Poker Championship, called KEmpfer. The poker bot played 1 million hands against itself and the observed pot sizes at the start of the turn round were recorded.
A second distribution of pot sizes was also created, based on the play of human players. For that 1 million hand histories of play between players at the stake of 0.5$\1$ on the online poker site PokerStars were analyzed. The following graph shows the distributions of pot sizes of both, human players and KEmpfer. The graph has been smoothed for a better visual representation.

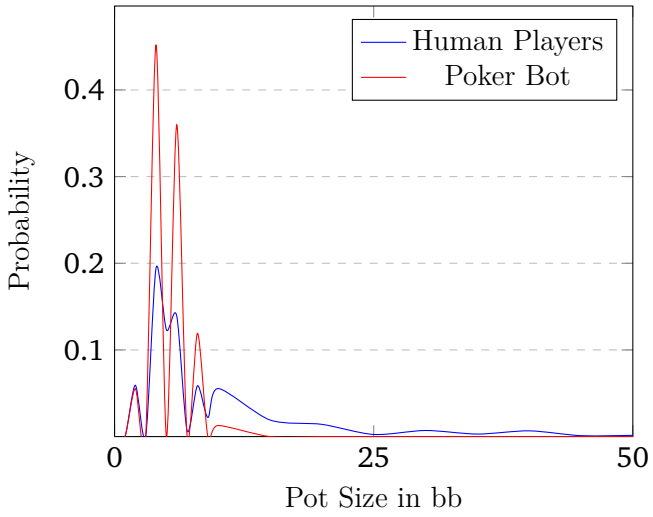


Figure 20: Probability Distribution of Pot Size

It was observed, that the play of the poker bot resulted in only few pot sizes at the start of the turn round, with a pot size of 4 having around 45% probability, a pot size of 6 having around 36% probability and a pot size of 8 having around 12% probability. The probability of one of the 3 most common pot sizes being observed was around 93%. The Annual Computer Poker Competition [ACPC] is played with a stack size of 200 big blinds, meaning, that pot sizes of up to 400 big blinds are possible, however pot sizes higher than 50 big blinds at the start of the turn had such a low probability, that they are not shown in the graph. The main reason for that distribution is, that the bot used an action abstraction with only few bet sizes and because he was playing against himself, no action translation ever took place. Observing the play of 2 poker bots with different action abstractions, which would also require action translation, might be preferable, because it would probably produce a more even distribution.
The play of human players resulted in more observed pot sizes, with a pot size of 4 being the most common with a probability of around 19% and a pot size of 5 having a probability of around 12%. Around 66% of the probability mass was contained in pot sizes between 2 and 10, with 2 being the smallest possible pot size on the turn due to the rules of the game, namely the posting of the blinds.
The probability distributions for KEmpfer and the human players can be found in the appendices.

Table 15: Most Common Turn Pot Sizes

	2bb	**3bb**	**4bb**	**5bb**	**6bb**	**7bb**	**8bb**	**9bb**	**10bb**	**Sum**
Poker Bot	5.54%	0%	45.16%	0%	35.98%	0%	11.92%	0%	1.28%	99.88%
Humans	5.92%	0%	19.43%	12.32%	13.90%	0.67%	5.86%	2.20%	5.54%	65.84%

There are potential problems using training data created by sampling the pot size according to any of the 2 distributions. The goal of a poker AI using an approximated Nash equilibrium as its strategy is to perform well against every possible strategy, not only against likely strategies. It is possible, that the neural network will not predict situations with an unusual pot size accurately, because it did not have much training data for those pot sizes. Especially big pot sizes are not sampled with a high probability, which might lead to big mistakes for a poker AI using these neural networks. As the pot is so big, a mistake could be even more costly, than in a situation with a small pot size.
On the other hand there are benefits to using a pot size distribution created from observed hands, instead of using a uniform distribution for example. The neural networks used by DeepStack used 10 million training examples and it was speculated by the author, that more training examples might have improved the accuracy even further, so using a uniform distribution might have produce to few examples for the most relevant situations.
Using the distribution created from the hand histories of human players might also improve the results of a poker bot against humans. Even if using that distribution might produce a program, that is potentially more exploitable, it might increase the performance in practice, as human players do not have the necessary informations to optimally exploit a poker AI.
Another reason to use a distribution as in observed hands played by a poker bot comes from the definition of a Nash equilibrium, since no player can increase his expectation by unilaterally deviating from a Nash equilibrium, a potential opponent would have to give up some expectation on early rounds in order to produce a situation with a unlikely pot size on the turn. This mistake might be big enough to compensate for any mistake of the poker AI resulting from suboptimal prediction of counterfactual values in this situation.

5.4.2 Mixed Distributions

Instead of either using a probability distribution based on observed data or using a uniform distribution, another approach might be to find a compromise. Pot sizes could be sampled from multiple distributions with different probabilities. For example a sampling method mixing the distributions of humans and earlier poker bots might result in a neural network which produces a strategy with a low exploitability, while still improving the results against humans to some degree. A distribution of observed pot sizes could also be mixed with a uniform distribution, creating a distribution which covers more possible bet sizes as shown in the following graph.

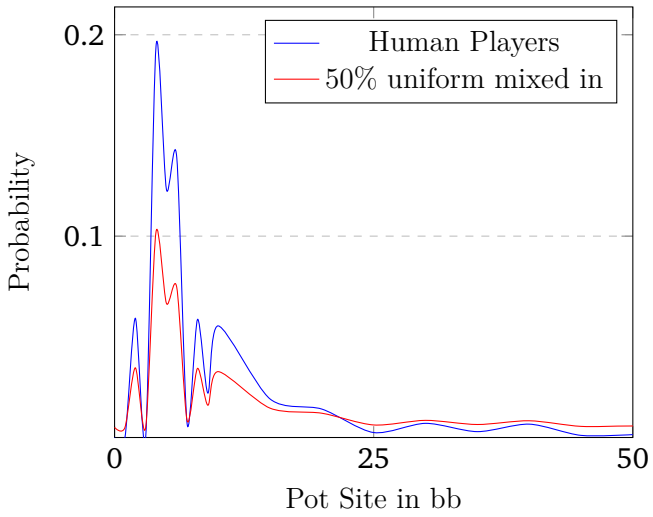


Figure 21: Mixed Probability Distribution

While the distribution of bet sized created by human players concentrated around 66% of the probability mass in the pot sizes between 2 and 10, a method which samples pot sizes from a uniform distribution half of the time results in only around 35% of combined probability for those pot sizes, making the network potentially more accurate in situations with bigger pots.

5.4.3 Analysis of Used Distribution

The training examples used for the main neural networks of this thesis are using the distribution of human pot sizes, without any other distributions mixed in. The reason for not using the distribution created by the play of KEmpfer is, that it uses too few pot sizes. It is unclear which bot was used by the University of Alberta in order to create their distribution, so the distribution created by KEmpfer might be highly different. While the human distribution also favors only a few pot sizes, it still uses more sizes than the distribution of the bot and even unlikely pot sizes still get sampled with a non-zero probability.
On the other hand it is interesting to see how well neural networks will predict counterfactual values for pot sizes, which were not sampled very often, which would not be possible with a uniform distribution or a mixed distribution with a high ratio of a uniform distribution. The rest of this section will analyze the effect of the pot size on the quality of the network's predictions.

Experimental Setup
In order to analyze the effect of the pot size sampling on the prediction error, a special test set was created, consisting of 5000 examples with a uniform pot size distribution.
Like the training examples of the baseline implementation, they were solved using 2000 iterations of CFR+, but a new pot size was sampled for every training example, making the algorithm slightly slower, but ensuring a more even distribution.
The baseline implementation, which was trained with a human pot size distribution, was then used in order to predict the counterfactual values of test examples with different pot sizes, in order to see, if pot sizes, which were not used often in training, produced a higher prediction error. The reason why an extra test set had to be created was, that some pot sizes were extremely rare on the original test set, which would lead to a very low statistical significance for some pot sizes.

Experimental Results
The following graph shows the distribution used in the training of the neural network and the Huber loss

on the new test set.

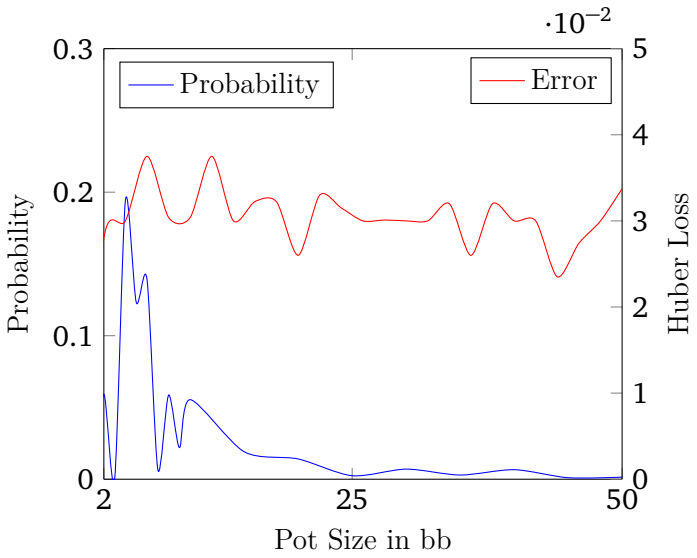


Figure 22: Huber Loss by Pot Size

The following table again shows the Huber loss for different turn starting pot sizes.

Table 16: Huber Loss by Pot Size

Pot Size	**2-5**	**6-10**	**11-15**	**16-20**	**21-25**	**26-30**	**31-35**	**36-40**	**41-45**	**46-50**
Huber Loss	0.041	0.031	0.030	0.032	0.032	0.030	0.032	0.031	0.032	0.031

We can not observe a clear pattern in which the starting pot size affects the Huber loss of a data set. Even though the network was trained with much fewer training examples with a big pot size, the Huber loss of test sets with a high pot size is very similar to test sets with a much more common, small pot size.
We will now further analyze how test examples with different starting pot sizes differ from one another. For the first test we will look at the average counterfactual values for different pot sizes, as in the original DeepStack implementation and our baseline implementation, they are expressed as a fraction of the pot. The following graph shows the average counterfactual values for different pot sizes

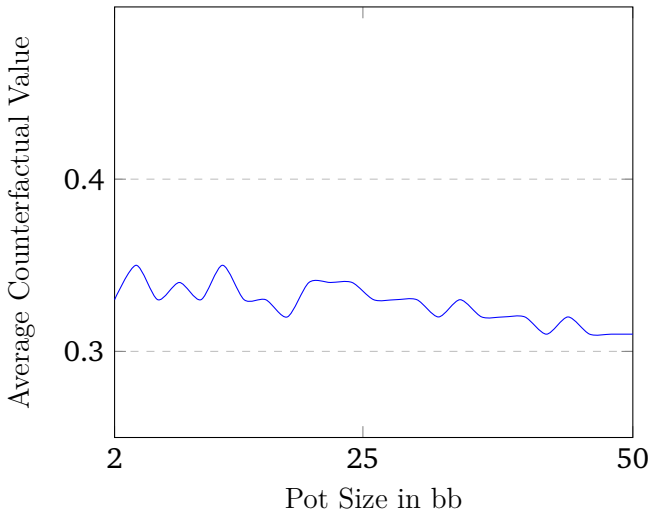


Figure 23: Average Counterfactual Values by Pot Size

We can see, that the average counterfactual values of the test set do not vary much depending on the pot size, with a pot size of 2 having the highest average of 0.349 and the pot size of 46 having the lowest average of 0.319. The reason for this is likely the fact, that the counterfactual values are expressed as a fraction of the pot size, suggesting an almost linear relation between the absolute counterfactual values and the starting pot size of turn sub games.
As a second step we will analyze the standard deviation of counterfactual values depending on the pot size. Even though the averages of counterfactual values are very similar for all pot sizes, datasets with a higher standard deviation might produce a higher loss, when a quadratic loss function is applied. The following graph shows the standard deviation of counterfactual values depending on the pot size

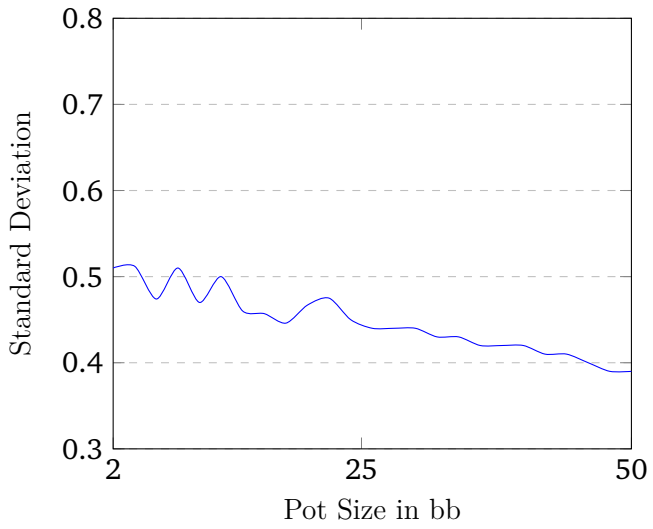


Figure 24: Standard Deviation by Pot Size

As the pot size increases, the standard deviation of the test data decreases slowly, with the pot size of 8 having the most variance in its counterfactual values with a standard deviation of 0.51 and a pot size 0f 50

having a the smallest standard deviation of 0.39. Expressing the counterfactual values as a fraction of the pot size does however again make the test data rather similar, as the pot size of 50, while being 25 times bigger than the pot size of 2, only has an about 20% lower standard deviation.
In order to minimize the effect of a quadratic loss function another test was conducted. We will again use all the available training data of the baseline implementation and create a neural network model, this time using the more intuitive mean absolute error. The model will then again be tested on the new test set described in this section, with 5000 test examples and a uniform distribution of pot sizes. The following graph shows the results of the test

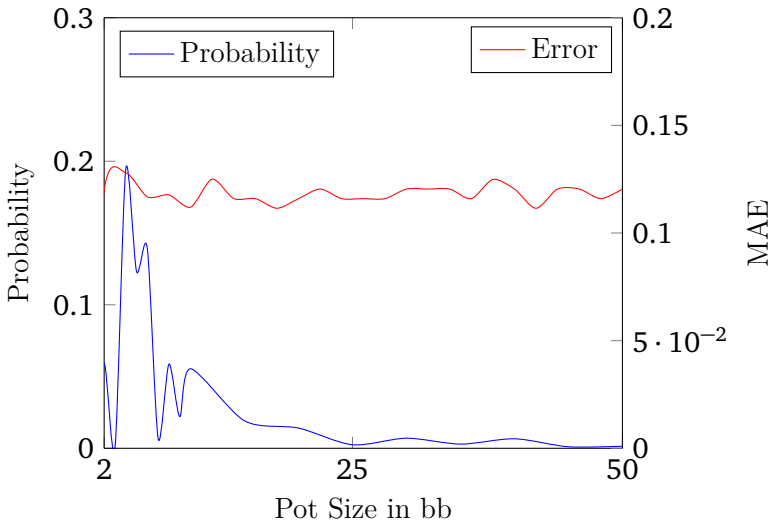


Figure 25: Mean Absolute Error by Pot Size

We can again not observe any substantial effect of the pot size on the prediction error.

5.4.4 Summary

In this section we have first described the starting pot size distributions based on the play of a Nash equilibrium based poker bot as well as human pot size distributions. We have then analyzed, how the pot size affects the prediction error of the model, especially for pot sizes, which were not sampled often during training. We found out, that the starting pot size of a test example does not substantially affect the Huber loss or the mean absolute error.
We have furthermore seen, that our test examples had very similar average counterfactual values as well as a similar standard deviation, this was probably mostly due to the fact, that the counterfactual values were represented as a fraction of the pot size, which makes turn solutions produced with different starting pot sizes very similar to each other.
The conclusion is therefore, that the pot size does not play a big role, for the accuracy of the predictions. This section only analyzed the model's prediction quality based on the pot size and only with the same counterfactual value representation, that DeepStack used, as a fraction of the pot size. We can not make any statements about in game performance. From a strategical viewpoint, a different representation might be better. Representing counterfactual values as absolute values and not as a fraction of the pot size might be beneficial, as it might prevent costly mistakes in bigger pots. In order to answer that question however, a full implementation of a poker bot using counterfactual value networks and continual resolving would have to be created, which goes beyond the scope of this thesis.

5.5 Number of Iterations

In this section the trade-off between the number of iterations used for the creation of training examples and the size of the resulting training set will be analyzed. Because the total available computational time for this thesis was fixed, any increase in the number of iterations per training example resulted in an anti-proportional decrease in the number of created training examples.
In chapter 4.2 the exploitability and the change of counterfactual values of training examples were analyzed directly, this section will look at the predictions of different neural network models in order to find the best number of iterations.

Experimental Setup
The test set for this experiment will consist of test examples which were trained using 2000 iterations, as those are likely to be closest to a real Nash equilibrium from all the created endgame solutions. We will use 20% of the 300,000 endgame solutions, which were created for the baseline implementation, for this purpose.
We will use 8 different training sets and compare their results on the aforementioned test set. As the number of iterations and the data set size are anti-proportional, given a fixed amount of computation time, the product of the number of iterations and the training set size will be equal for all training sets. The training sets will be created from only those 80% of endgame solutions, which are not part of the test set. The minimum number of iterations tested will be 600, as we want to use a reasonable number of training examples for training sets with a higher number of iterations. By choosing this minimum number of iterations, even the training set with 2000 iterations will still use 30% of the maximum training set size.

Experimental Results
The following graph shows the Huber loss of neural network models, which were trained on training sets with different sizes and numbers of iterations. All values refer to an unabstracted test

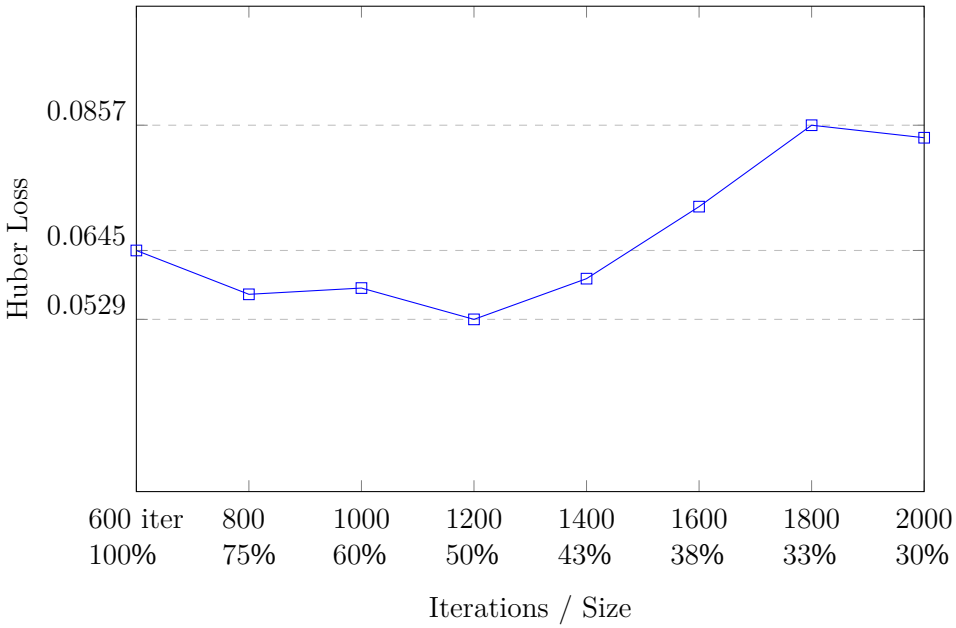


Figure 26: Huber Loss by Iterations and Data Set Size

Using a training set which was created with 1200 iterations of the CFR+ solver and used 50% of the maximum training set size produced the best result with a Huber loss of 0.0529. The training set created

with 1800 iterations, using 33% of the maximum training set size produced the worst result with a Huber loss of 0.0857.
In general a bigger training set size produced better results, than using more iterations, which we can see at the examples of 600 iterations with a Huber loss of 0.0645 and 2000 iterations with a Huber loss of 0.0836. We have however, chosen a minimum of 600 iterations, which already produced a decent level of convergence, for an extremely low number of iterations the results might look differently.
We might have assumed, that the prediction error, as a function of the number of iterations used for the training set, would resemble a convex function, with a single local minimum also being the global minimum. We can however observe, that the error increases from 800 to 1000 iterations for example and then falls again. It is unclear however, if this is only a result of insufficient data.
We have obtained all results from a test set, which used endgame solutions trained for 2000 iterations, as this is the test set, that is likely to be the closest to a real Nash equilibrium, from all the data, that was produced during the work on this thesis. The test set does however still not represent a perfect Nash equilibrium of the game and more accurate results could probably be achieved, by using an even higher number of iterations for the test set and maybe a finer grained action abstraction.

6 Distribution Encoding

This chapter will attempt to find alternative encodings for input player distributions and otput counterfactual values to DeepStack's potential aware card abstraction. We will analyze the encoding error of each abstraction as well as the prediction error on the abstracted and unabstracted test set.
After examining several abstracted encodings, we will analyze an unabstracted encoding, which introduces no encoding error.
The architecture of all networks constructed in this chapter will be equal to the network used by DeepStack, except for the input and output layers, which will be described for each encoding.

6.1 $E[HS^2]$ Abstraction

In this section we will analyze an encoding based on the $E[HS^2]$ card abstraction, which was described in chapter 3. The abstraction used 1326 equal distance buckets, the possible $E[HS^2]$ values, ranging from zero to one, were divided into 1326 regions and all hands whose $E[HS^2]$ values fell into the same region, were assigned to the same bucket. For this kind of abstraction only an $E[HS^2]$ lookup table has to be created, there are no further steps necessary, like clustering of hands.
The number of buckets of 1326 was chosen for two reasons, first of all it is reasonably close to the number of buckets which DeepStack used, which was 1000, the second reason was, that it is the number of private card combinations, which means, that in the best case each hand might be assigned to a separate bucket. In reality however, some $E[HS^2]$ values will not be present on certain boards, because there is no hand with that specific $E[HS^2]$ value on that board. This means, that the number of buckets used on each turn will be lower than 1326, the following chart shows the distribution of the number of buckets present on each board

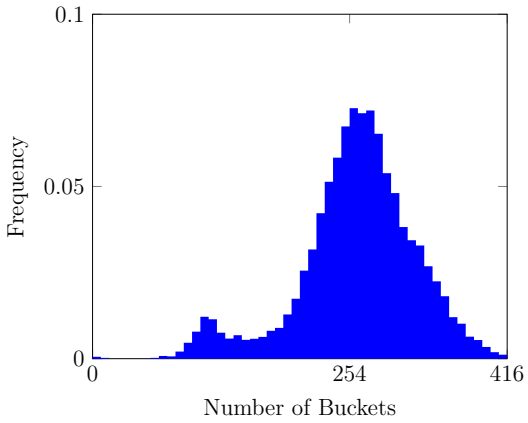


Figure 27: Distribution of Number of Buckets per Turn

The number of buckets per turn is much higher for the $E[HS^2]$ abstraction than for the potential aware abstraction analyzed in the last chapter, with an average number of buckets per turn of 254.36, compared to 14.47.

Table 17: Number of Buckets per Turn

Min	**Max**	**Mean**	**Median**
3	413	254.36	258

While the $E[HS^2]$ abstraction uses more buckets on each turn, resulting in a finer differentiation of private card information, it does not capture any public card information. This might lead to a higher encoding error in practice. The following table shows the encoding errors of the $E[HS^2]$ abstraction, compared to the potential aware card abstraction, on our data set

Table 18: Encoding Error

	E[HS2]	**Potential Aware**
Encoding Error (Huber Loss)	0.0240	0.0258
Encoding Error (MSE)	0.0509	0.0544

We can see, that the $E[HS^2]$ abstraction introduces a smaller encoding error, than the potential aware card abstraction, although not by a big margin.
After analyzing the encoding error, we will now compare the quality of predictions, produced by neural networks using both abstractions. The results are listed in the following table

Table 19: Prediction Error

	E[HS2]	**Potential Aware**
Error on Abstracted Training Set (Huber Loss)	0.0254	0.0052
Error on Unabstracted Training Set (Huber Loss)	0.0387	0.0267
Error on Abstracted Test Set (Huber Loss)	0.0330	0.0102
Error on Unabstracted Test Set (Huber Loss)	0.0434	0.0297

Even though the $E[HS^2]$ abstraction had a lower encoding error, the potential aware card abstraction outperformed it in terms of the accuracy of the neural networks. The potential aware abstraction performed better in its own abstraction, as well as after mapping the counterfactual values of buckets back to counterfactual values of cards.

6.2 Nested Public Card Abstraction

In this section we will analyze a nested public card abstraction similar to the abstraction described in chapter 3, using the **draw value** and **highcard value** of the board and later subdividing the public buckets using private card information. We will start by describing the features by which boards are classified and then presenting 2 possible encodings based on those features. We will then compare both encodings in order to highlight a general concept.

Draw Value
A **draw** is typically regarded as a hand which currently only has a small chance of winning at a showdown, but might improve to a strong hand after certain public cards are revealed on later rounds. Usually mostly draws which can improve to a straight or a flush are considered, as those draws have the highest probability to improve and straights and flushes are some of the strongest hands possible. We will consider any hand, that can be a straight or a flush on the last round to be a draw, regardless of hand strength on the current

betting round. In order to calculate the draw value of a board, we will roll out all possible future public cards and count the number of possible straights and flushes, the draw count will then be normalized to a value between zero and one.

```
TurnDrawValue(board);
Input:
        Turn Board board
Data:
Sum of future Straights and Flushes drawCount;
for each riverCard in allRiverCards do
    for each holeCards in allHoleCards do
        if !overlaps(board, riverCard, holeCards) then
            rank = rankHand(board, riverCard, holeCards);
            if rank == FLUSH or rank == STRAIGHT then
                drawCount++
            end
        end
    end
end
normalize(drawCount)
return drawCount
```

Algorithm 3: Turn Draw Value

Highcard Value

The second feature used is the high card value of a board. Human experts will often use different strategies on boards with many high cards such as kings and aces compared to boards with mostly cards of a low rank. The reason for that being that many pairs with a low rank will have a low hand strength on these boards, making them likely to fold to a bet, which can lead to many players using those boards for bluffing. A second reason is, that a player who raised on the first betting round usually has a high chance of having many high cards in his hand distribution, which means that he will usually want to play very differently on boards with high cards, which improve many of his hands to pairs, than on boards with mostly low cards.

```
TurnHighcardValue(board);
Input:
        Turn Board board
Data:
Highcard Value highcardValue;
for each turnCard in board do
    highcardValue += rankof(turnCard);
end
normalize(highcardValue)
return highcardValue
```

Algorithm 4: Turn Highcard Value

Preclustered Card Abstraction

Based on the 2 presented features describing public board cards, we can create card abstractions which in turn can be used to encode a counterfactual value network's inputs and outputs. The first abstraction that we will be presenting is the preclustered public card abstraction.

In order to create the abstraction we have first clustered public boards into 10 buckets based on their draw and highcard values, using the k-means algorithm with the Euclidian distance. The public board buckets were then subdivided into 100 $E[HS^2]$ buckets, resulting in a total of 1000 buckets, the same number of buckets, that DeepStack used. The following figure shows the distribution of the public buckets.

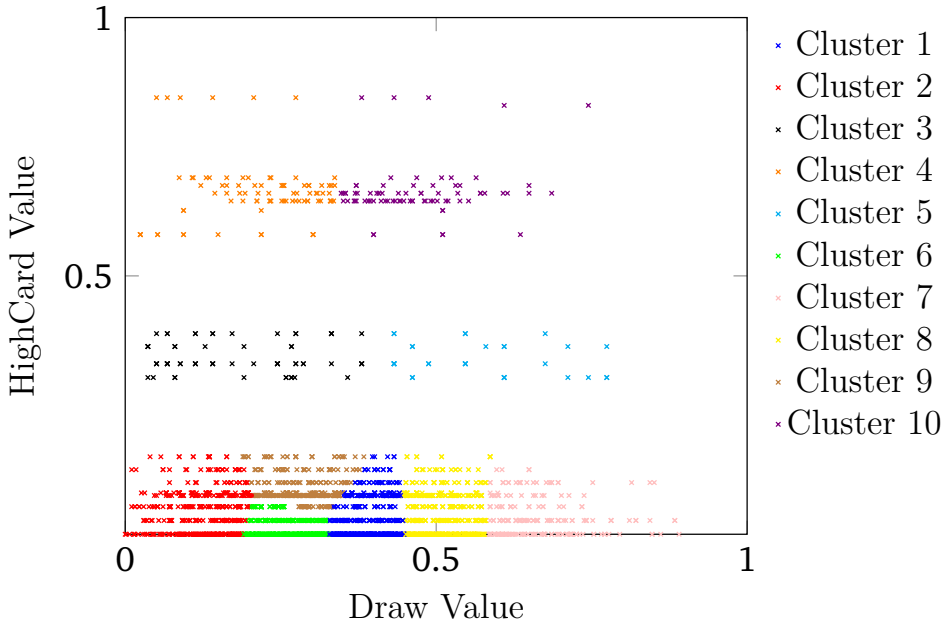


Figure 28: Public Board Clusters

Following the definition of the card abstraction we know, that the number of buckets per turn will be 100, the following table shows the encoding error of the preclustered public card abstraction compared to the potential aware card abstraction

Table 20: Encoding Error

	Public Nested (Preclustered)	**Potential Aware**
Encoding Error (Huber Loss)	0.0406	0.0258
Encoding Error (MSE)	0.0886	0.0544

From the table we can see, that while the nested public card abstraction uses more buckets on each turn, making the encoding potentially better, the actual encoding error on the constructed training set is higher. The nested public card abstraction seems not to reflect the strategical properties of poker hands as well as the potential aware card abstraction, which DeepStack used, thus mapping strategically different hands to the same bucket. This intuition is also supported by the prediction error of the networks trained with both abstractions, as shown in the next table.

Table 21: Prediction Error

	Public Nested (Preclustered)	**Potential Aware**
Error on Abstracted Training Set (Huber Loss)	0.0080	0.0052
Error on Unabstracted Training Set (Huber Loss)	0.0436	0.0267
Error on Abstracted Test Set (Huber Loss)	0.0161	0.0102
Error on Unabstracted Test Set (Huber Loss)	0.0478	0.0297

Just like with the $E[HS^2]$ abstraction, the potential aware abstraction outperforms the public nested abstraction on every data set.

Using Public Board Features Directly

A different way of encoding the distribution of the players as well as the public board structure, is to use the draw and high card values directly as inputs of the neural network. This has the goal of shifting more of the responsibility to the network, instead of a pre-built card abstraction based on expert knowledge.
Strictly speaking, this is no longer a nested card abstraction, as no boards are mapped to buckets, which are later subdivided, but instead the high card and draw values are used directly as inputs. The only real bucketing in this encoding is the private card $E[HS^2]$ bucketing, however conceptually it is similar to a nested public card abstraction and it was therefore chosen to describe it in the same section.
For the first network using this technique, private hands were first bucketed into 100 $E[HS^2]$ buckets in order to represent the private cards distribution and the draw and high card values were fed directly into the network. Compared to the abstraction used in the last section this has two potential benefits, the first being that by feeding the public board features directly into the network, it can potentially generalize better. The second benefit is, that the total number of inputs is lower, while all information from the last network is still available, although in a different form.
The second network will use the same direct input of draw and high card values, but will use 1000 $E[HS^2]$ buckets, making the number of inputs comparable to the preclustered nested public card abstraction. The following table shows the results of both abstractions

Table 22: Prediction Error

	Public Nested (100 E[HS2])	**Public Nested (1000 E[HS2])**	**Public Nested (Preclustered)**	**Potential Aware**
Error on Abstracted Training Set (Huber Loss)	0.0183	0.0254	0.0080	0.0052
Error on Unabstracted Training Set (Huber Loss)	0.0484	0.0386	0.0436	0.0267
Error on Abstracted Test Set (Huber Loss)	0.0268	0.0334	0.0161	0.0102
Error on Unabstracted Test Set (Huber Loss)	0.0522	0.0438	0.0478	0.0297

Conclusions
The first result is, that none of the public nested card abstractions performed better than the potential aware card abstraction. Both abstractions are similar in the way, that they capture private card information as well as public board information, but the potential aware abstraction does it in a more precise way. We can however observe, that while using 100 private card buckets and feeding the draw and highcard value of a board directly into the neural network resulted in a higher error, than using the preclustered abstraction, the non clustered abstraction with 1000 buckets performed better. In the case of the public nested abstraction doing less preprocessing resulted in a better result, the next section will test an encoding, which uses no abstraction.

6.3 No Abstraction

We will now look at an encoding, which is using no lossy card abstraction.
For the input card distributons all 1326 possible private card combinations will be considered, and the distributions will be encoded as a vector of 1326 probabilities of holding that combination.
The board cards will be encoded as a vector of 4 integers ranging from 0 to 51, representing one of the 52 cards in the deck.
The idea behind the encoding is to give the neural network the possibility to find the relation between inputs and outputs completely on its own, without any potentially lossy preprocessing.
Because each possible hand is represented individually, meaning that no strategically different hands are mapped to the same bucket, there is no direct encoding error. The following table shows the prediction error on the used data sets

Table 23: Prediction Error

	Unabstracted	**Potential Aware**
Error Abstract Training Set (Huber Loss)	0.0254	0.0052
Error Unabstracted Training Set (Huber Loss)	0.0254	0.0267
Error Abstract Test Set (Huber Loss)	0.0301	0.0102
Error Unabstracted Test Set (Huber Loss)	0.0301	0.0297

Because the unabstracted encoding introduces no encoding error, there is no difference between the abstracted and unabstracted training set, as well as the abstracted and unabstracted test set. We can see, that the unabstracted encoding achieves a Huber loss of 0.0301 on its test set, which is close to the error of the potential aware abstraction, which is 0.0297. Furthermore the unabstracted encoding performs better, than all abstraction based encodings, except the potential aware abstraction.
Because of the promising results of the unabstracted encoding, a different lossless encoding was tested. The second unabstracted encoding, that we will consider, will again represent the player hand distributions as a vector of 1326 probabilities, but a one hot encoding will be used for the public board cards. For each board card a vector of 52 inputs was used, representing again the 52 cards in the deck, if the card was present on the public board, the input was set to one, otherwise it was zero. The following table shows the prediction error of the encoding

Table 24: Prediction Error

	Unabstracted (One Hot)	**Potential Aware**
Error Abstract Training Set (Huber Loss)	0.0102	0.0052
Error Unabstracted Training Set (Huber Loss)	0.0102	0.0267
Error Abstract Test Set (Huber Loss)	0.0143	0.0102
Error Unabstracted Test Set (Huber Loss)	0.0143	0.0297

The unabstracted one hot encoding was able to substantially reduce the prediction error compared to the previously described unabstracted encoding. The Huber loss on both, the training set and the test set decreased by over 50%. Furthermore, while the potential aware encoding was able to produce a lower Huber Loss in its own abstraction, the unabstracted one hot encoding outperformed the abstraction on the unabstracted training set and the unabstracted test set. The unabstracted one hot encoding was therefore better than the potential aware encoding at predicting counterfactual values of actual hands instead of buckets, which is the most important measure in actual game play.

6.4 Summary

This section first tested neural network input and output encodings based on traditional card abstractions. While the $E[HS^2]$ card abstraction produced a lower encoding error, using the same number of buckets, the predictions of the neural network were less precise, than those of the potential aware abstraction.
The public nested card abstraction had a higher encoding error, than the potential aware card abstraction and the predictions were also worse, we could however observe, that increasing the number of private buckets and feeding features describing the board directly into the neural network improved the accuracy.
Motivated by the results of feeding a neural network less pre-processed inputs, encodings using no lossy card abstraction was tested. The first encoding, representing the board cards as a vector of 4 integers, achieved results comparable to those of the potential aware card abstraction. The second encoding, using a one hot encoding for the board cards, was then able to achieve a lower prediction error, than the potential aware abstraction.

7 Endgame Based Abstraction

In this chapter we will present a new card abstraction technique based on the turn sub-game solutions returned by the CFR+ turn solver. While this example will focus on the application for poker, the technique is not domain specific and can be used for state space abstraction in any imperfect information game which can be divided into a trunk and an endgame as described in chapter 3.
We will create two card abstractions based on this concept, the resulting card abstractions will then be tested by using them as a feature for the encoding of deep counterfactual value networks, similar to the experiments conducted in chapter 6. The abstractions are however not limited to the use as an encoding feature, but can be used in any way, that earlier card abstractions can be used, for example in order to solve an abstracted game using CFR, as described in chapter 3.

7.1 Counterfactual Value Based Abstraction

The first presented card abstraction is based on the intuition, that cards which have similar counterfactual values in a sub-game starting from a certain round, are also strategically similar on said round.
In order to create the card abstraction, a solution for each of the 16432 canonical turn boards [K. Waugh, 2013] has to be computed, this will automatically also ensure a solution for each canonical private hand. A uniform hand distribution was chosen for each player as well as a fixed pot size of 10 big blinds. After solutions were computed for all canonical turns, we then traversed the game tree of each solution and recorded the counterfactual values for every hand in different states in a feature vector, as shown in Figure 29.

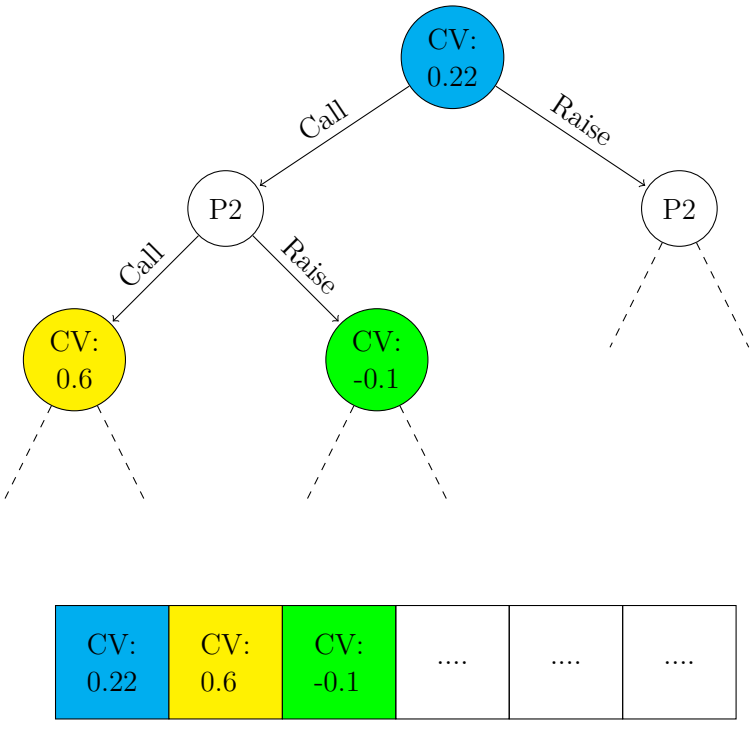


Figure 29: Creating Feature Vector from solved Game Tree Counterfactual Values

The traversal strategy can be chosen arbitrarily, as long as it is consistent for all training examples. The canonical hands are then clustered using k-Means, according to their counterfactual value feature vector.

Due to computational limitations, for this thesis only the counterfactual values of the first and second player at the root of the game were considered, resulting in a feature vector of length 2. The following table shows the encoding error compared to the potential aware card abstraction

Table 25: Encoding Error

	Endgame Based (CV)	**Potential Aware**
Encoding Error (Huber Loss)	0.0531	0.0258
Encoding Error (MSE)	0.1212	0.0544

Unfortunately the results were disappointing, as the Huber loss and MSE of the endgame based encoding are about twice as high as those of the potential aware encoding.
We will now look at the prediction error of the resulting neural network model, which is shown in the following table

Table 26: Prediction Error

	Endgame Based (CV)	**Potential Aware**
Error on Abstracted Training Set (Huber Loss)	0.0051	0.0052
Error on Unabstracted Training Set (Huber Loss)	0.2159	0.0267
Error on Abstracted Test Set (Huber Loss)	0.0100	0.0102
Error on Unabstracted Test Set (Huber Loss)	0.2313	0.0297

Just like the encoding error, the results of the neural network trained using the endgame based abstraction are substantially worse than the results of the potential aware card abstraction.

7.2 Strategy Based Abstraction

The second card abstraction is based on the intuition, that cards, which have similar strategies in a sub-game starting from a certain round, are also strategically similar on said round.
For this abstraction again a solution has to be created for each canonical turn, as this was already done for the counterfactual value based card abstraction, the same data could be reused.
As described in chapter 3, a strategy for a given information set is a function assigning a probability to each legal action in that information set. For each private hand we can traverse the game tree of a turn sub-game solution and store the probabilities of each action in the order in which they occur. Any traversing scheme can be used, as long as it is consistent for all hands. The strategies in each state of the solved game tree can then again be used to create a feature vector, as shown in the following Figure

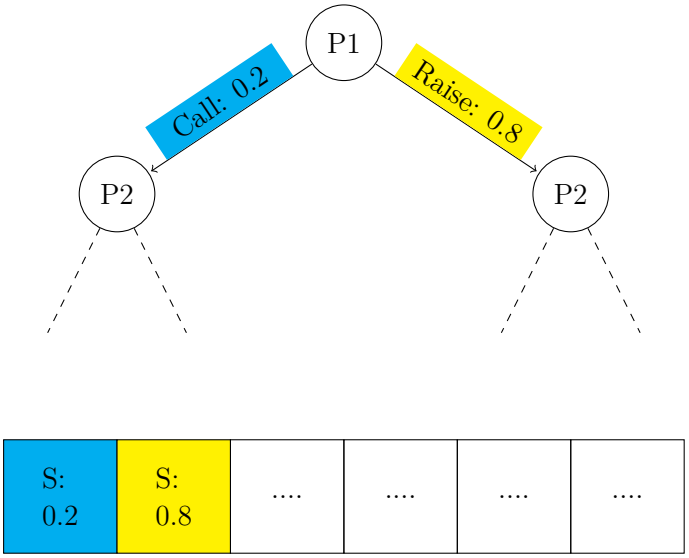


Figure 30: Creating Feature Vector from solved Game Tree Strategy

Due to computational limitations only the strategy of player 1 in the root of the game tree, as well as the first action of player 2 after the first player's check were considered. Because the solutions of the endgames used the fcpa action abstraction and the fold action is not a legal action in both states, this resulted in 3 action probabilities for each player and a feature vector of overall size 6. The following table shows the encoding error of the strategy based endgame abstraction compared to the potential aware abstraction

Table 27: Encoding Error

	Endgame Based (Strat)	**Potential Aware**
Encoding Error (Huber Loss)	0.0622	0.0258
Encoding Error (MSE)	0.1291	0.0544

Similar to the counterfactual value based endgame abstraction, the strategy based endgame abstraction produced a higher encoding error, than the potential aware card abstraction. The following tables shows the results of the resulting neural network model

Table 28: Prediction Error

	Endgame Based (Strat)	**Potential Aware**
Error on Abstracted Training Set (Huber Loss)	0.0049	0.0052
Error on Unabstracted Training Set (Huber Loss)	0.3626	0.0267
Error on Abstracted Test Set (Huber Loss)	0.0099	0.0102
Error on Unabstracted Test Set (Huber Loss)	0.4902	0.0297

Unfortunately the endgame based abstraction was again not able to produce better results than the potential aware card abstraction. One thing, that we can observe, is, that there seems to be very little correlation between the error of an encoding on its abstract training and test set and the unabstracted results. The strategy based endgame abstraction performed very poorly on the unabstracted sets, yet it was better in its own abstraction, than the potential aware encoding. The likely explanation is, that a bad card abstraction loses a lot of differentiation between hands and creates an oversimplified data set in its encoding, which is easier to fit.

7.3 Summary

In this section we have presented new card abstractions based on the counterfactual values and strategies of endgame solutions. Unfortunately both abstractions performed substantially worse, than the card abstraction, which DeepStack used. For a side by side comparison of both endgame based abstractions and the potential aware abstraction, the results are again presented in the following table

Table 29: Prediction Error

	Endgame (Strat)	**Endgame (CV)**	**Potential Aware**
Error on Abstracted Training Set (Huber Loss)	0.0049	0.0051	0.0052
Error on Unabstracted Training Set (Huber Loss)	0.3626	0.2159	0.0267
Error on Abstracted Test Set (Huber Loss)	0.0099	0.0100	0.0102
Error on Unabstracted Test Set (Huber Loss)	0.4902	0.2313	0.0297

While the endgame based card abstraction could be used in every way, that other, traditional card abstractions could be used, it appeared to be an especially good abstraction for the encoding of counterfactual value network inputs and outputs. A card abstraction based on counterfactual values of endgame solutions seemed to be a good method to group card counterfactual values into buckets, but in the experiments, which were conducted, this intuition could not be confirmed.
One reason for the poor performance of the abstraction might be the fact, that a uniform hand distribution for both players does not represent situations with different hand distributions well enough. Another problem might have been, that, due to computational limitations, only a small part of the solved trees was

considered. The endgames could also have been solved for multiple starting pot sizes, which would have resulted in a longer feature vector, which might have led to better results.
The potential aware encoding remains therefore the best abstraction based encoding, which was analyzed in this thesis, only the unabstracted one hot encoding performed better than the original DeepStack method.

8 Conclusions and Outlook

In this thesis we have analyzed the techniques behind the strong computer poker AI called DeepStack, with a focus on its deep counterfactual value networks, which are used for the prediction of the counterfactual values of future rounds.

We have first described techniques for improving the run time of the CFR+ endgame solver, which is necessary for the creation of training examples for deep counterfactual value networks. For the technique of fast terminal node evaluation, which was published in an earlier paper by a different author, pseudo code was provided. Furthermore ways of saving time by reducing memory allocation and by using look up tables for turn all-in situations were described. All techniques combined resulted in a run time decrease of 89% over the baseline algorithm.

We have then analyzed the way in which DeepStack encodes its input private card distributions and its counterfactual value outputs. We have found out, that by encoding the inputs and outputs using the potential aware card abstraction, the algorithm introduces a potential problem, which we called implicit card abstraction. We have then focused on the encoding of only the outputs and discovered, that the abstraction reduces the accuracy of the training data, which we called the encoding error. We have also analyzed the distribution of buckets which are used by each training example and identified the problem of sparse inputs and undefined outputs, which required us to use a weight matrix for the loss function of the neural network optimizer.

We have then argued, that additionally to implicit card abstraction, creating the training examples for DeepStack using the fcpa action abstraction introduces a form of implicit action abstraction. For this we have presented theoretical arguments and compared the results of solutions using the fcpa action abstraction to finer grained abstractions empirically.

DeepStack has to sample a random pot size for each training example, it does that by following a distribution of turn starting pot sizes derived from the play of earlier Nash equilibrium based poker bots. We have examined the pot size distribution based on the play of a Nash equilibrium based poker bot, which was created for an earlier project, and a distribution of pot sizes resulting from the play of human poker players. We found out, that with both distributions many pot sizes get sampled with a very small probability, which might produce inaccurate predictions for situations with those pot sizes. That hypothesis was tested, but no increase in the prediction error was found for unlikely pot sizes. This finding seems to be the result of the encoding of counterfactual values in the training and test data as fraction of the pot, which makes the data very similar for all pot sizes.

As computational resources are usually limited, we have then analyzed the trade-off between the number of iterations used for the creation of training examples and the size of the data set, by comparing the prediction error on a test set created with the maximum number of iterations. A number of iterations close to that of the original DeepStack implementation performed best in our tests. We have to keep in mind, that, while the used test set is likely to be the closest to a real Nash equilibrium, from all the test sets we could have created with limited computational resources, it still does not represent a real Nash equilibrium. This test could be more accurate if more iterations and maybe also a finer grained card abstraction were used for the test set, as this would be even closer to a Nash equilibrium of the real game.

In an attempt to improve the encoding of DeepStack's inputs and outputs, several techniques based on traditional card abstractions were analyzed. While the $E[HS^2]$ abstraction was able to produce a lower encoding error than DeepStack's potential aware card abstraction, the resulting neural network model was less accurate than the one using the potential aware abstraction. All other abstractions performed worse in terms of both, the encoding error, as well as the prediction error.

After testing abstraction based encodings, two encodings, which used no lossy card abstraction, were tested. While the first encoding was only able to produce similar results as the potential aware card abstraction, the second unabstracted encoding, using a one hot representation of the board cards, performed better than the abstraction on the training and test set.

Finally two new card abstractions, based on the solutions of end games, were presented. The first card

abstraction was based on the counterfactual values of a hand in the end game, the second on the strategy of the hand. Unfortunately the results were poor, as the abstractions had high encoding end prediction errors.
The most important result of this work seems to be the results of the unabstracted one hot encoding of deep counterfactual value network inputs and outputs. In the future other lossless encoding techniques could be analyzed, those techniques might for instance make use of isomorphisms for a better generalization.
In comparison to the original DeepStack implementation, the implementation for this thesis unfortunately used a much smaller data set, due to computational limitations. In the future some of those experiments could be conducted with more data.
It was also not possible to implement continual re-solving during the work on this thesis due to time constraints, which means the results could not be tested in poker game play. It might be interesting to build a poker bot in the future, which uses a different neural network than DeepStack and compare the results.

References

O. Tammelin, N. Burch, M. Johanson, and M. Bowling, "Solving Heads-up Limit Texas Hold'em", in: *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

H. W. Kuhn, "A Simplified Two-Person Poker", in: *Contributions to the Theory of Games 1*, pp. 97-103, 1950.

M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, "Regret minimization in games with incomplete information", in: *Proceedings of the Annual Conference on Neural Information Processing Systems(NIPS)*, 2007.

J. F. Nash, "Non-cooperative games" in:*Annuals of Mathematics*, 1951

M. Johanson, N. Bard, M. Lanctot, R. Gibson, and M. Bowling, "Efficient Nash Equilibrium Approximation through Monte Carlo Counterfactual Regret Minimization" in:*Autonomous Agents and Multiagent Systems 2012 (AAMAS-12)*, 2012

R. Gibson "Regret Minimization in Games and the Development of Champion Multiplayer Computer Poker-Playing Agents", PhD Thesis, 2014.

O. Tammelin "Solving Large Imperfect Information Games Using CFR+" in:*arXiv preprint arXiv:1407.5042*, 2014

M. Johanson "Robust Strategies and Counter-Strategies: Building a Champion Level Computer Poker Player", MSc Thesis, 2007.

D.P. Schnizlein "State Translation in No-Limit Poker", MSc Thesis, 2009.

A. Gilpin, T. Sandholm and T.B. Sorensen "A heads-up no-limit Texas Holdem poker player: Discretized betting models and automatically generated equilibrium-finding programs" in:*Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems (AAMAS-08)*, 2008

N. Brown and T. Sandholm "Libratus: The Superhuman AI for No-Limit Poker" in:*Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, 2017

M. Bowling, N. Burch, M. Johanson and O. Tammelin "Heads-up limit holdem poker is solved" in:*Science*, 2015

Annual Computer Poker Competition, "http://www.computerpokercompetition.org/"

M. Johanson "Measuring the Size of Large No-Limit Poker Games" in:*Technical Report TR13-01, Department of Computing Science, University of Alberta*, 2013

M. Moravcik, M. Schmid, N. Burch, V. Lisy, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson and M. Bowling "DeepStack: Expert-level artificial intelligence in heads-up no-limit poker" in:*Science*, 2017

K. Waugh, M. Zinkevich, M. Bowling and M. Johanson, "Accelerated Best Response Calculation in Large Extensive Games", in: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence(IJCAI)*, 2011.

K. Waugh, "A Fast and Optimal Hand Isomorphism Algorithm" in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013)*, 2013.

M. Johanson, N. Burch, R. Valenzano and M. Bowling, "Evaluating State-Space Abstractions in Extensive-Form Games", in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems(AAMAS 2013)*, 2013.

M. Moravcik, "Evaluating public state space abstractions in extensive form games with an application in poker ", MSc Thesis, 2014.

D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron, "Approximating game-theoretic optimal strategies for full-scale poker", in: *International Joint Conference on Artificial Intelligence, pages 661668*, 2003.

A. Gilpin and T. Sandholm, "A competitive texas holdem poker player via automated abstraction and real-time equilibrium computation", in: *National Conference on Artificial Intelligence*, 2006.

M. Johanson, N. Bard, N. Burch, and M. Bowling, "Finding optimal abstract strategies in extensive-form games", in: *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence (AAAI-12)*, 2012.

K. Waugh, D. Schnizlein, M. Bowling, and D. Szafron, "Abstraction pathology in extensive games", in: *Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2009.

John Hawkin, Robert C. Holte, and Duane Szafron, "Using Sliding Windows to Generate Action Abstractions in Extensive-Form Games", in: *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence (AAAI-12)*, 2012.

E. Jackson, "Slumbot NL: Solving Large Games with Counterfactual Regret Minimization Using Sampling and Distributed Processing", in: *Computer Poker and Imperfect Information: Papers from the AAAI 2013 Workshop (AAAI-13)*, 2013.

S. Ganzfried and T. Sandholm, "Endgame Solving in Large Imperfect-Information Games", Carnegie Mellon University, in: *Proceedings of the 14th International Conference on Autonoumous Agents and Multiagent Systems (AAAMAS 2015)*, 2015.

N. Burch, M. Johanson and M. Bowling, "Solving Imperfect Information Games Using Decomposition", in: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI-14)*, 2014.

Torch Library, "http://www.torch.ch/"

TensorFlow Library, "http://www.tensorflow.org/"

M. Moravcik, M. Schmid, N. Burch, V. Lisy, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson and M. Bowling "Supplementary Materials for DeepStack: Expert-level artificial intelligence in heads-up no-limit poker", 2017

T. Kanungo and D. Mount "An Efficient k-Means Clustering Algorithm: Analysis and Implementation", 2002

D. Arthur and S. Vassilvitskii, "k-means++: the advantages of careful seeding", in: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 2007

C. Elkan, "Using the Triangle Inequality to Accelerate -Means ", in: *Proceeding ICML'03 Proceedings of the Twentieth International Conference on International Conference on Machine Learning, pages 147-153*, 2003

M. Bowling, "Artificial Intelligence Goes All-In: Computers Playing Poker", [Video], *URL:https://www.deepstack.ai/*, 2017

Appendices

CreateTurnEquities(*board*);
Input:
Vector of Board Cards *board*;
Data:
Matrix of hand vs. hand equities *equities[1326][1326]*;
Current index in equity matrix for player$_0$ *index*$_0$;
Current index in equity matrix for player$_1$ *index*$_1$;

```
for c0 = 0 to 51 do
    for c1 = c0 + 1 to 51 do
        index1 = 0
        for c2 = 0 to 51 do
            for c3 = c2 + 1 to 51 do
                equities[index0][index1] = HandVsHandEquity(c0, c1, c2, c3, board);
                index1++;
            end
            end
        end
        index0++;
        end
        return equities
```

Algorithm 5: Create Turn Equities

HandVsHandEquity(*c0,c1,c2,c3,board*);
Input:
First card of player$_0$ c_0;
Second card of player$_0$ c_1;
First card of player$_1$ c_3;
Second card of player$_1$ c_4;
Vector of Board Cards *board*;
Data:
Sum of hands *hands*;
Sum of wins for player$_0$ *wins*;

```
for river = 0 to 51 do
    if !overlalps(board,river, c0, c1, c2, c3) then
        r0 = rankHand(board,c0,c1);
        r1 = rankHand(board,c2,c3);
        if r0 > r1 then
            wins++;
        end
        if r0 == r1 then
            wins += 0.5;
        end
        hands++;
    end
end
return wins / hands;
```

Algorithm 6: Hand vs. Hand Equity

Fold(*pHoles,oHoles,op,payoff*);
Input:
Distribution of trained player *pHoles[1326]*;
Opponent distribution *oHoles[1326]*;
Opponent reach probabilities *op[1326]* ;
Payoff from perspective of player$_0$ *payoff* ;
Data:
Sum of of fold probabilities *foldSum*;
Vector of utilities *u[1326]*

```
pValue = trainPlayer == 0 ? payoff : -payoff;
for i = 0 to 1326; do
    foldSum = 0;
    for j = 0 to 1326; do
        if !pHoles[i].overlaps(oHoles[j]) then
            foldSum += op[j];
        end
    end
    u[i] = foldSum * pValue;
end
return u
```

Algorithm 7: Fold Evaluation in $O(n^2)$

FastFold(*pHoles,oHoles,op,payoff*);
Input:
Distribution of trained player *pHoles[1326]*;
Opponent distribution *oHoles[1326]*;
Opponent reach probabilities *op[1326]* ;
Payoff from perspective of player$_0$ *payoff* ;
Data:
Vector of card removed opponent fold probabilities *cr[52]*;
Sum of fold probabilities *foldSum*;
Vector of utilities *u[1326]*

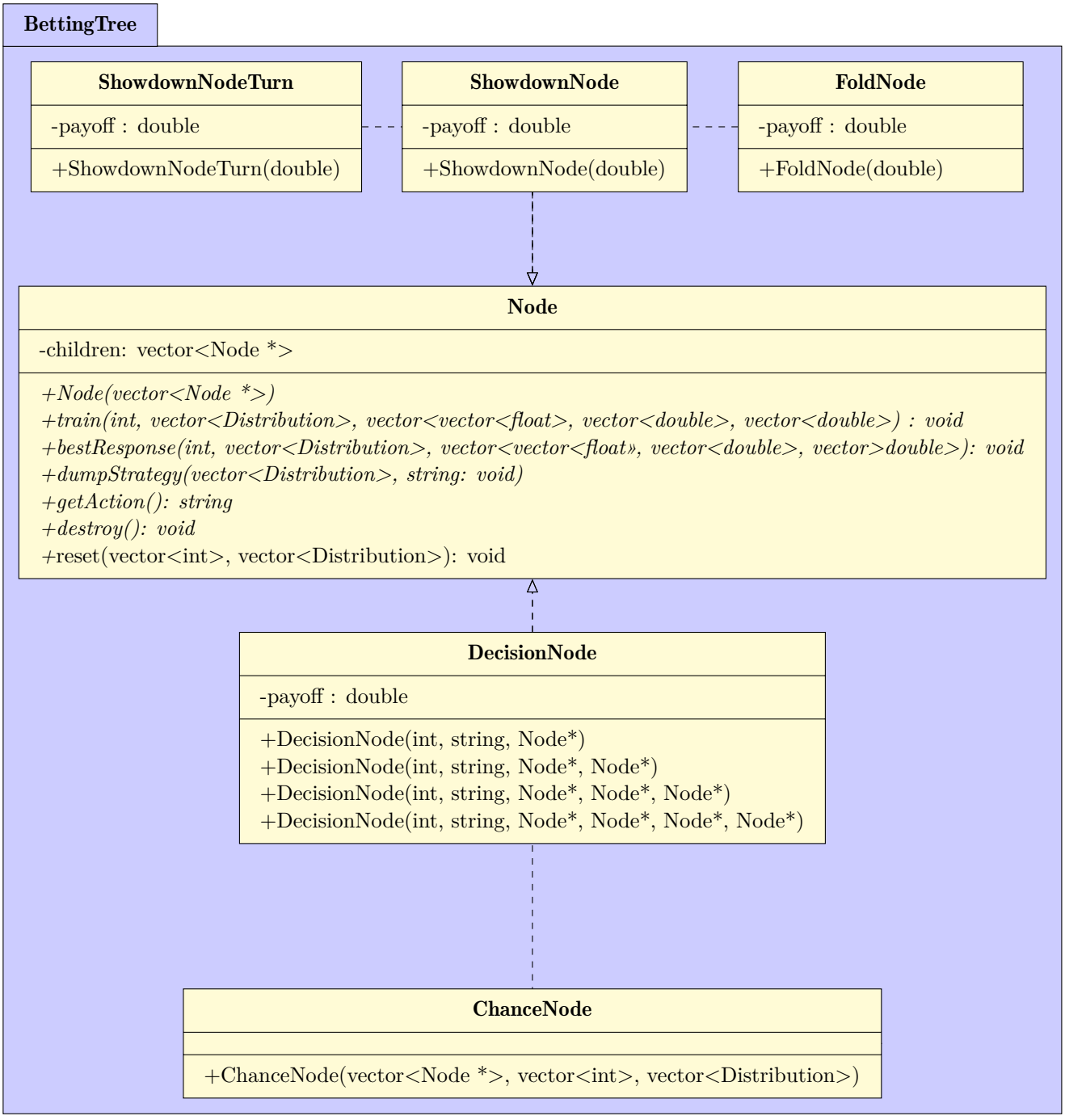
```
pValue = trainPlayer == 0 ? payoff : -payoff;
for i = 0 to 1326; do
    cr[oHoles[i].card0] += op[i];
    cr[oHoles[i].card1] += op[i];
    opSum += op[i];
end
for i = 0 to 1326; do
    u[i] = (opSum - cr[pHoles[i].card0] - cr[pHoles[i].card1] + op[i]) * pValue;
end
return u
```

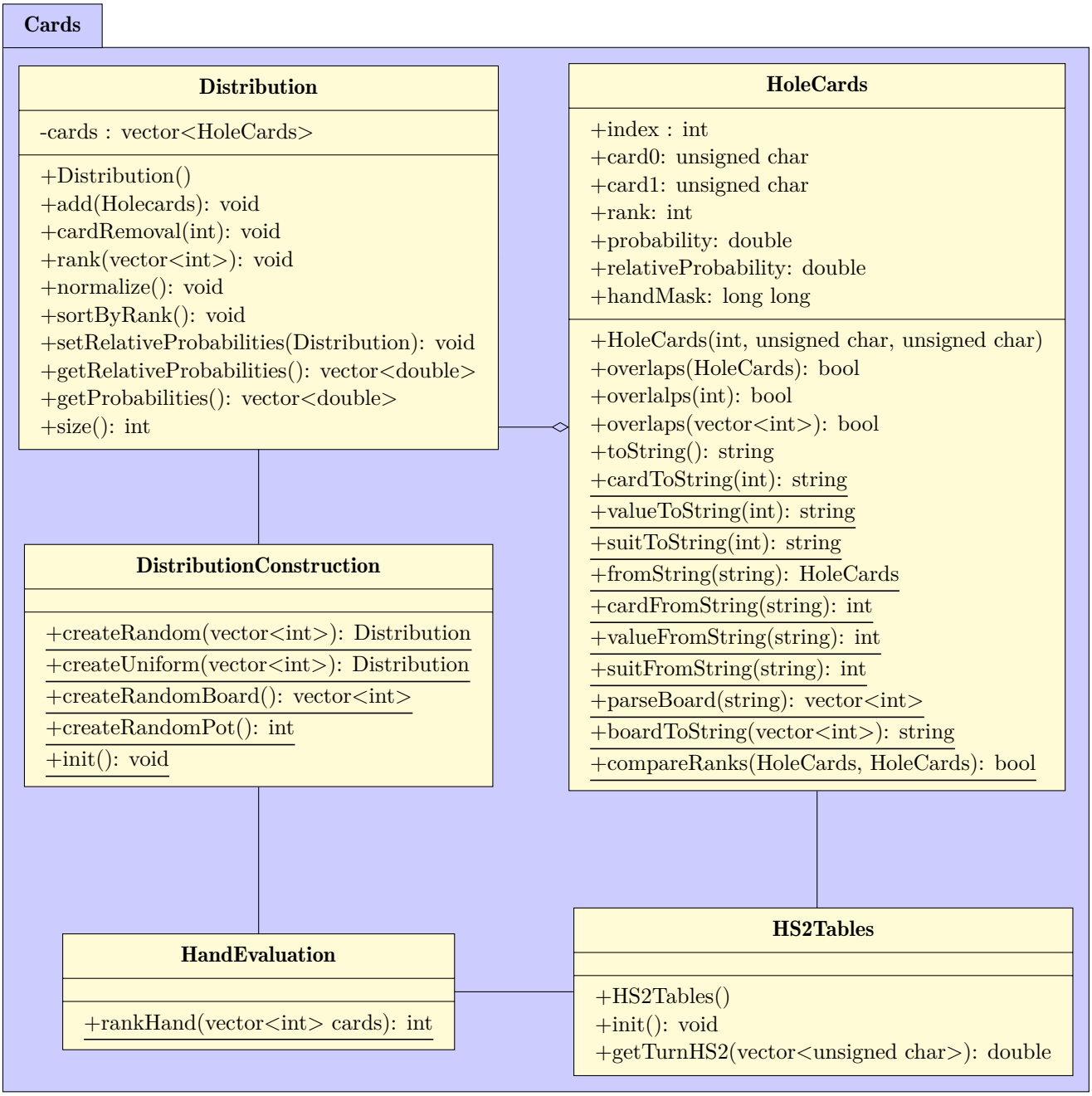
Algorithm 8: Fast Fold Evaluation in $O(n)$

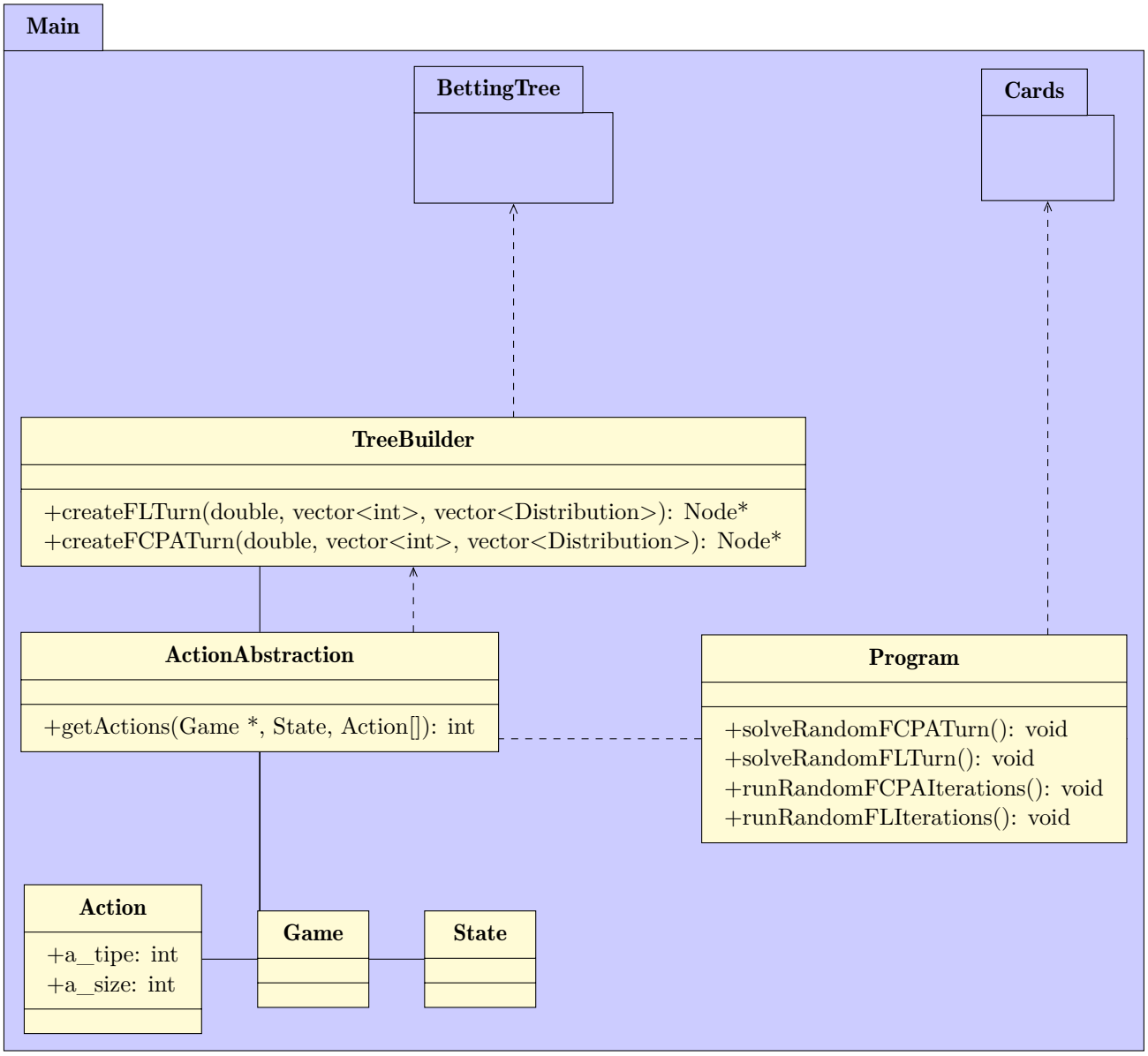
C Pot Size Distributions

PotSize	*Humans*	*Bot*	*PotSize*	*Humans*	*Bot*
0	0.000000000000	0.000000000000	50	0.001446536300	0.000000000000
1	0.000000000000	0.000000000000	51	0.000410473880	0.000000000000
2	0.059228968000	0.055433230000	52	0.001086548600	0.000000000000
3	0.000000000000	0.000000000000	53	0.001040452500	0.000000000000
4	0.194386820000	0.451687280000	54	0.000801192360	0.000000000000
5	0.123258500000	0.000000000000	55	0.000329257130	0.000000000000
6	0.139021140000	0.359881580000	56	0.000588272760	0.000000000000
7	0.006762941400	0.000000000000	57	0.000188774100	0.000000000000
8	0.058629720000	0.119276370000	58	0.000561932160	0.000000000000
9	0.022049252000	0.000000000000	59	0.000498275800	0.000000000000
10	0.055431534000	0.012802185000	60	0.000702415200	0.000000000000
11	0.025159635000	0.000000000000	61	0.000169018660	0.000000000000
12	0.062631294000	0.000868801700	62	0.000430229320	0.000000000000
13	0.012511771000	0.000000000000	63	0.000122922660	0.000000000000
14	0.035630010000	0.000048266764	64	0.000487300540	0.000000000000
15	0.019140815000	0.000000000000	65	0.000272185900	0.000000000000
16	0.017560380000	0.000002298417	66	0.000329257130	0.000000000000
17	0.010463792000	0.000000000000	67	0.000129507810	0.000000000000
18	0.018348401000	0.000000000000	68	0.000318281900	0.000000000000
19	0.003512076000	0.000000000000	69	0.000144873130	0.000000000000
20	0.014111960000	0.000000000000	70	0.000381938270	0.000000000000
21	0.002379431600	0.000000000000	71	0.000193164190	0.000000000000
22	0.005246164000	0.000000000000	72	0.000460959970	0.000000000000
23	0.003382568200	0.000000000000	73	0.000120727615	0.000000000000
24	0.009118227000	0.000000000000	74	0.000359987780	0.000000000000
25	0.002456258300	0.000000000000	75	0.000085606850	0.000000000000
26	0.004111324000	0.000000000000	76	0.000285356200	0.000000000000
27	0.002133586200	0.000000000000	77	0.000100972190	0.000000000000
28	0.006029795400	0.000000000000	78	0.000232675040	0.000000000000
29	0.003795237300	0.000000000000	79	0.000079021710	0.000000000000
30	0.007035127400	0.000000000000	80	0.000298526460	0.000000000000
31	0.001870180500	0.000000000000	81	0.000061461330	0.000000000000
32	0.006038576000	0.000000000000	82	0.000151458280	0.000000000000
33	0.002607716500	0.000000000000	83	0.000074631615	0.000000000000
34	0.005090315400	0.000000000000	84	0.000206334470	0.000000000000
35	0.002915023100	0.000000000000	85	0.000050486095	0.000000000000
36	0.007386335000	0.000000000000	86	0.000133897900	0.000000000000
37	0.002721859000	0.000000000000	87	0.000037315807	0.000000000000
38	0.003964256000	0.000000000000	88	0.000158043420	0.000000000000
39	0.002069929800	0.000000000000	89	0.000070241520	0.000000000000
40	0.006633433500	0.000000000000	90	0.000122922660	0.000000000000
41	0.002923803400	0.000000000000	91	0.000041705902	0.000000000000
42	0.003171843700	0.000000000000	92	0.000118532570	0.000000000000
43	0.001606774700	0.000000000000	93	0.000046096000	0.000000000000
44	0.004418630600	0.000000000000	94	0.000079021710	0.000000000000
45	0.001055817800	0.000000000000	95	0.000037315807	0.000000000000
46	0.003237695200	0.000000000000	96	0.000116337520	0.000000000000
47	0.001281907800	0.000000000000	97	0.000028535618	0.000000000000
48	0.002102855600	0.000000000000	98	0.000096582095	0.000000000000
49	0.001071183200	0.000000000000	99	0.000017560380	0.000000000000

D Turn Solver UML




Cards
Distribution
-cards : vector<HoleCards>
+Distribution()
+add(Holecards): void
+cardRemoval(int): void
+rank(vector<int>): void
+normalize(): void
+sortByRank(): void
+setRelativeProbabilities(Distribution): void
+getRelativeProbabilities(): vector<double>
+getProbabilities(): vector<double>
+size(): int
HoleCards
+index : int
+card0: unsigned char
+card1: unsigned char
+rank: int
+probability: double
+relativeProbability: double
+handMask: long long
+HoleCards(int, unsigned char, unsigned char)
+overlaps(HoleCards): bool
+overlalps(int): bool
+overlaps(vector<int>): bool
+toString(): string
+cardToString(int): string
+valueToString(int): string
+suitToString(int): string
+fromString(string): HoleCards
+cardFromString(string): int
+valueFromString(string): int
+suitFromString(string): int
+parseBoard(string): vector<int>
+boardToString(vector<int>): string
+compareRanks(HoleCards, HoleCards): bool
DistributionConstruction
+createRandom(vector<int>): Distribution
+createUniform(vector<int>): Distribution
+createRandomBoard(): vector<int>
+createRandomPot(): int
+init(): void
HandEvaluation
+rankHand(vector<int> cards): int
HS2Tables
+HS2Tables()
+init(): void
+getTurnHS2(vector<unsigned char>): double


Main
BettingTree
Cards
TreeBuilder
+createFLTurn(double, vector<int>, vector<Distribution>): Node*
+createFCPATurn(double, vector<int>, vector<Distribution>): Node*
ActionAbstraction
+getActions(Game *, State, Action[]): int
Program
+solveRandomFCPATurn(): void
+solveRandomFLTurn(): void
+runRandomFCPAIterations(): void
+runRandomFLIterations(): void
Action
+a_tipe: int
+a_size: int
Game
State