ADAPTIVE POKER BOT BASED ON PLAYER-MODELLING

Bjørn Hasager Vinther & Nicolai Guldbæk Holst

bhas@itu.dk ngul@itu.dk

A thesis submitted for the degree of

Bachelor Softwaredevelopment

May 2014

Abstract

Nothing yet...

Preface

This bachelor thesis consists of 15 ETCS points. The thesis has been written exclusively by Nicolai Guldbæk Holst and Bjørn Hasager Vinther, both studying *Software development* at the IT University of Copenhagen. It has been written in the spring semester 2015. Our supervisor was Kasper Støy.

Introduction

Poker is arguably the most popular card game in the world. It involves intelligence, statistics, psychology, and luck. Even though the game itself is fairly simple, it takes years of practice to master all elements of the game.

In poker reading an opponent refers to the process of figuring out the opponents strategy based on their actions. The top players are able to read their opponents and adapt to their strategy in order to gain an advantage. Players often mix in unorthodox actions or change strategies in order to mislead the opponents.

Another big element of poker is statistics. Since the players do not know the cards that will be dealt throughout the game, players must calculate the likelihood of winning.

The goal of this thesis is to develop an adaptive poker computer (APC), that is capable of adapting to the strategies of the opponents. The APC will be programmed to play a variation of poker called Texas hold'em limit poker. For the scope of this thesis the APC will only play against a single opponent, also called heads-up in poker.

For the readers unfamiliar with Texas hold'em limit poker the following section describes the basic rules and flow of the game.

Texas hold'em poker

This section only describes the basics of Texas hold'em limit poker. For a more in-depth description see [**?**].

Poker is played with a standard deck of 52 cards. This thesis only concerns Texas Hold'em limit poker. [**?**].

Game play

A Texas hold'em poker game consists of multiple rounds. In each round each player is dealt two private cards called hole cards that are hidden for the opponents. Five public cards, called community cards, that is visible for everybody are dealt as the round progresses.

A round is divided into four game states: pre-flop, flop, turn, and river. Each game state starts with cards being dealt and then the bidding begins. In the pre-flop the dealer will deal the hole cards to each player. In the flop the first three out of five community cards are dealt and in the turn and river the fourth and fifth community card are dealt respectively.

If only one player is left after the bidding that player wins the pot, otherwise the game continues to the next game state. If multiple players are still left after the bidding succeeding the river, a showdown will start. During showdown each player still in the game will reveal their hole cards and a winner will be found. The winner wins the pot. In case of a draw the pot is split between the winners.

The amount of chips is the main indicator of how well the player is doing. The main goal for the players is to increase their amount of chips.

The bidding

The bidding is where the players in turn perform their actions. The actions include: call, bet, raise, fold, check, all-in. The player to the left of the dealer is always the first to act.

A player can play aggressively by betting or raising which will increase the cost for the other players. Likewise a player can also play defensively by calling or checking which will not increase the cost for other players. If a player decides to fold he will lose what he betted that round. Whenever a player chooses to play aggressively all other players must either fold or call in order for the bidding round to stop. The bidding continues until all players have called the aggressor or folded.

Rules for determining the winner

The rules for finding the best hand a quite simple. Each player has to create the best possible hand choosing five of the seven card available cards (hole cards and the community cards). Each playing card has a card rank (2, ... ,

Q, K, A) and a suit (diamond, heart, club, or spade). The possible ranks of a hand can be seen in table 1.

First the rank of the hand of every player is found. In case two players has the same rank, the winner will be determined by more advanced rules. These rules is not described in this section as they are irrelevant for this thesis. An case the hands are still even after using the advanced rules the round results in a draw and the pot is split between the players.

Starting from the bottom (the worst hand) of table 1 we have the explanations:
High card is the highest card.
One pairs is having two cards with the same card rank.
Two pairs is having two pairs
Three of a kind is having three cards with the same card rank
Straight is having five cards in a row (e.g 10-A or 4-9)
Flush is having five cards with the same suit
Full house is having three of a kind and a pair
Four of a kind is having four cards with the same card rank
Straight Flush is the same as a royal flush except there is no requirement for the highest card.
Royal Flush is having a straight all in the same suit. Furthermore the highest card have to be an ace.

rank	name	example hand
1	Royal flush	A♣ K♣ Q♣ J♣ 10♣
2	Straight flush	7♣ 6♣ 5♣ 4♣ 3♣
3	Four of a kind	K♣ K♠ K♢ K♡ 10♣
4	Full house	K♣ K♠ K♢ Q♡ Q♣
5	Flush	K♡ Q♡ 5♡ 3♡ 2♡
6	Straight	A♣ K♠ Q♢ J♡ 10♣
7	Three of a kind	A♣ A♢ A♠ J♣ 10♣
8	Two pairs	A♣ A♢ 5♠ 5♣ 4♣
9	One pair	A♣ A♡ J♣ 9♠ 2♡
10	High card	A♣ K♢ 6♡ 5♡ 3♡

Table 1: Ranks of different hands in Texas hold'em poker sorted best to worst.

Artificial intelligence and poker

In the field of artificial intelligence, games are interesting because of their well-defined game rules and success criteria. Computers have already mastered some of the popular games, one example is the chess computer Deep Blue which won against Garry Kasparov, the world champion of chess at the time. Chess is a game of perfect information, as no information is hidden from the players.

Since then the interest of the artificial intelligence research has shifted towards games with imperfect information. These types of games presents new challenges such as deception and hidden information. Poker is an example of a game with imperfect information.

In may 2015 the contest *Brains Vs. Artificial Intelligence* [12] was held with four of the best poker players in the world. Each of the players played ∼20.000 hands of Texas Hold'em no-limit heads-up against Claudico, the world's best poker computer at the time. Claudico was able to beat one of the four players, which proves that artificial intelligence in regards to poker

have come a long way.

Developing an algorithm capable of playing poker is not only limited to the domain of poker, but can end up having a future applications in other domains as well.

> *"Bowling says the findings in this new research are especially valuable because they give us a hint at the scale of problems AI can solve. … Solving a game as complex as heads-up limit Texas hold 'em could mean a breakthrough in our conception of how big is too big"*

- Lily Hay Newman, *Using AI to Study Poker Is Really About Solving Some of the World's Biggest Problems* - Line 46-52

In essence a game presents a challenge for the player to solve. How the player approaches this challenge and what strategy the player uses to solve it, is what determines the players success.

In order to achieve the goal of developing an APC, we will try to answer the following problem statements:

Problem statements

1. How can one determine the strength of a poker hand?
2. How can we develop a default strategy without having information about the opponents?
3. How can one develop a strategy adapted to the opponents.

Our thesis is divided into three chapters each of which focuses on one of the three problem statements.

In chapter 1 we find a solution to problem statement one. We develop a subsystem which is able to estimate the probability of winning for any set of hole cards in any poker state. The subsystem can calculate the probability of winning with an error percentage of one percent and it takes less than a second on average.

In chapter 2 we try to answer problem statement two. We implement a self-learning algorithm using artificial neural networks and use it to observe data from real-life poker games in order to learn the strategies of the players. The self-learning algorithm did not learn the strategy of the players. The data observed by the algorithm turned out not to be suited for the algorithm.

In chapter 3 ...

Contents

1 Determine the strength of a poker hand

Our first step towards developing an adaptive poker bot is to find a way to determine the strength of any given hand in any game state. In this section we will answer the following problem statement:

Problem statement 1

How can one determine the strength of a poker hand?

The strength of a hand reflects the probability of winning so the stronger a hand is the more likely one are to win. Since the player does not know the outcome of the community cards during a round of poker, the player has to calculate the probability of winning based on the possible outcomes of the community cards. It is too time consuming to check every outcome as there are more than 250 millions different outcomes of community cards alone. In order to find the probability of winning without having to check all possible outcomes, one can instead make an estimate rather than calculating the true probability.

1.1 Design

When trying to estimate the strength of a hand, two options exist.

The first option is to create a simplified formula. Such formulas already exist, but since they are very simple they tend to be rather inaccurate. This option is straight forward, but the disadvantage is, that it is hard to make a formula that is accurate for every game state.

The second option is to use the Monte Carlo method to simulate a large amount of games and get the distribution of outcomes. This distribution can then be used to find the probability for any of the outcomes.

For a human player a simplified formula is necessary but due to the computational power of a computer the Monte Carlo method is optimal for the APC. The computer can perform thousands of simulations in no time. This method also gives a trade-off between accuracy and the number of simulations. This allows one to adjust the accuracy of the probability by adjusting the number of simulations. The major poker sites also use the Monte Carlo method to determine each players probability of winning.

The solution is implemented as a subsystem that uses the Monte Carlo method. We will refer to this subsystem as the calculator.

Algorithm 1: Pseudo-code for a single simulation

```
Data: Hole cards, number of opponents, community cards (optional)
Result: win = 1, draw = 0, lose = -1
Random missing community cards;
Determine rank of players hand;
foreach opponent do
    Determine rank of opponents hand;
    if opponent has stronger hand then
        return -1;
    end
    else if opponent has same hand then
        return 0;
    end
end
return 1;
```

The calculator takes three arguments: the hole cards of the player, the number of opponents, and the community cards (optional). The calculator simulates a pre-defined number of rounds and returns an object containing the distribution of wins, draws, and loses.

The calculator considers it a win only if the hand beats every other hand of the opponents.

In order to ensure the quality of the calculator, we have the following requirements:

- It must have a maximum error percentage of one percent. (deviation from the true probability)
- It must calculate the probability in less than five seconds.
- It must be able to return the probability of winning with a given hand in any game state with up to ten players.

1.1.1 Monte Carlo method

The Monte Carlo method is used to find the distribution of outcomes for a domain. Given a set of user defined inputs the Monte Carlo method performs the simulation to find the outcome. For each simulation the method tracks the outcomes and as the number of simulations increases, so will the accuracy of the distribution. This distribution can help to get a better understanding of the domain.

1.2 Test

We have created tests to ensure the calculator fulfils all the requirements.

In order to find the accuracy of the probabilities we first need to know the true probabilities. For this we use caniwin **??**. Caniwin is a website that has simulated all possible outcomes of any pre-flop hands. Caniwin's results are limited to the pre-flop game state, so we can only test the calculator for this game state.

The number of simulations affects the error percentage as well as the calculation time. We start by finding a number of simulations that meets the first and second requirement. We test what number of simulations are needed in order to get a maximum error of one percent. Afterwards we will find the number of simulations that results in a calculation time in less than five seconds.

Once we have a suitable number of simulations we test if the calculator finds the correct probability.

1.2.1 Finding the maximum error

As mentioned earlier the number of simulations affects the accuracy of the result. In this test we will find the number of simulations that fulfils the requirement of having a maximum error percentage of one percent.

Each test is performed with the hand J♣ J♢ in pre-flop with one opponent. The calculator calculates the probability 50 times for each test. The true probability found by caniwin is ∼77,1 %. We then calculate the error of each calculation using the the formula:

$$err = P_O - P_T$$

Here P_O is the probability found by the calculator and P_T is the true probability.

We plotted the results of each test in a graph, see figure 1, 2, and 3. Each test result is indicated with a red dot. The three graphs clearly shows that when using a higher amount of simulations, the more accurate the probability are.

Table 2 shows the range of results as well as the maximum error for the tests.

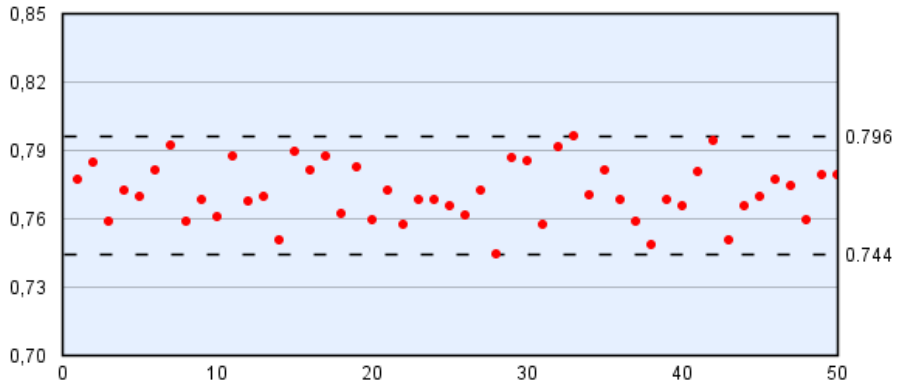

Figure 1: Result of the calculator with 1000 simulations

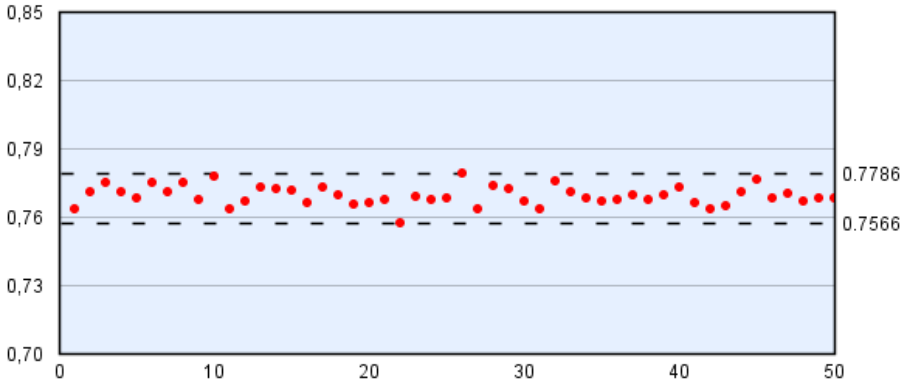

Figure 2: Result of the calculator with 10.000 simulations

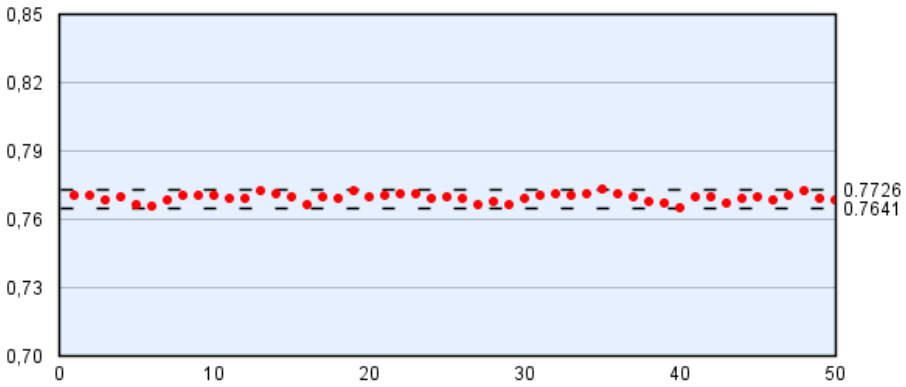

Figure 3: Result of the calculator with 50.000 simulations

simulations	range (%)	max error (%)
1000	5,2	2,7
10.000	2,2	1,5
50.000	0,9	0,7

Table 2: Combined test results from running the calculator with different numbers of simulations.

Using 50.000 simulations fulfils our requirements.

1.2.2 Finding the calculation time

In this test we run the calculator 100 times with the same hand in order to find an average computation time. In table 3 the average computation time can be seen. As we expect the computation time increases as the number of simulations increases.

simulations	Average computation time
1000	∼0,01 seconds
10.000	∼0,04 seconds
50.000	∼0,15 seconds

Table 3: Calculation times from running the calculator with different numbers of simulations.

Luckily, using 50.000 simulations gives a average computation time of ∼0,15 seconds which also satisfies the second requirement. We therefore settle for 50.000 simulation as the number of simulations for the calculator.

1.2.3 Determining the correctness of the calculator

To test if the calculator can calculate the correct probabilities, we find the probability of a number of different pre-flop scenarios and compare the result with the true probability. Every test is preformed with 50.000 simulations against one opponent. We use the formula described in section 1.2.1:

$$err = P_O - P_T$$

The result can be seen in table 4. From the results we can see that for all tests the error percentage is less than one and the calculator therefore works for the pre-flop.

hole cards	P_O (%)	P_T (%)	error (%)
A♣ A♢	85,2	84,9	0,3
8♣ 8♢	67,9	68,7	0,8
Q♣ k♣	62,8	62,4	0,4
A♡ 8♠	58,8	60,5	0,7
J♠ Q♢	57,2	56,9	0,3
10♡ J♡	56,7	56,2	0,5
3♢ 3♠	53,0	52,8	0,2
2♢ 2♡	49,5	49,4	0,1
9♢ 3♠	37,8	37,4	0,4
2♢ 7♢	35,5	35,4	0,1
2♢ 7♡	31,9	31,7	0,2

Table 4: Test results for different hole cards in pre-flop with one opponent.

Testing the calculator for other game states is a bit problematic because we were not able to find any probabilities for anything but pre-flop. Instead we calculate the probability of some combinations of hole cards and community cards.

In table 5 we can see the results. For the first situation (A♣ K♢) the player start with a strong hand but does not hit anything from the community cards. As expected the probability decreases. The second hand (2♢ 5♠) is bad until the turn where it hits a straight. This is also clearly shown by the percentages. Likewise the third hand (8♠ 9♠) hits a flush on the river and the probability increases drastically. Finally we have the hand (5♠ 5♣) where there is a straight from the community cards. This results in a zero percent win chance because the community cards are public and therefore the result is either a draw or a lose. The data suggests that the calculator also works for all game states.

hole cards	community cards	pre-flop (%)	flop (%)	turn (%)	river (%)
A♣ K♢	3♠ 7♣ T♣ 8♢ 2♠	64,5	54,4	41,7	35,6
2♢ 5♠	3♠ 6♢ K♢ 4♡ 7♠	29,7	28,2	92,8	94,2
8♠ 9♠	2♢ K♠ 3♠ A♣ T♠	49,1	53,5	38,5	99,7
5♠ 5♣	A♢ K♡ Q♠ J♢ T♡	58,5	48,0	34,4	0,0

Table 5: Test results for different hole cards in pre-flop, flop, turn, and river with one opponent

opponents	1	2	3	4	5	6	7	8	9
win probability (%)	85,2	73,4	64,0	56,1	49,5	43,9	39,1	35,0	31,6

Table 6: Chance of winning with the hole cards A♠ A♣ in pre-flop

From table 6 we can see that the probability also decreases as the number of players increases, which suggest it works for multiple players as well.

The calculator passed all tests and seems to work in every situation.

1.3 Discussion

For the implementation of the calculator the Monte Carlo method was. This method works well for the needs. The calculator can calculate the probability with an error percentage of less than one percent when using 50.000 simulations. This solution can be used for any poker state with up to ten players. At first it did not seem necessary to optimise the calculator any further, as the running time is acceptable. But in regards to later use or if were to scale the project, we decided to make an obvious optimisation by making it multi-threaded which speeded up the calculations by roughly 3 times.

Since we compare the results of the calculator to caniwin, one would have to ensure the reliability of that source. It is not directly shown who have written the article but when trying to contact the owner of the write we are directed to Bobby. It is very difficult to know if Bobby is the author and

whether or not he has any reputation and expertise in the field. However after finding out that the probability from caniwin heads-up and 10 players is very close to the results from the calculator we concluded that caniwin were a reliable source of information.

Alternatively we could have created our own formula to calculate a rank, or used an existing one, for instance the Chen formula. By using this method we will get a less accurate result but it will be easier to calculate. We would also have to create a formula for each game state which would cause even more work. Since performance is not a problem for our calculator we chose to use the Monte Carlo method.

1.4 Conclusion

In this section we have answered the question:

Problem statement 1

How can we predict the probability of ending up with the winning hand?

We have implemented a subsystem called the calculator that can estimate the probability of winning with a set of hole cards. The calculator uses the Monte Carlo method and it works for every poker state with up to ten players. We have found that 50.000 simulations is a good number of simulations for our calculator.

We compare our results to caniwin, which is a website that calculates the actual probabilities of all the 169 combinations of hole cards. The calculator has a maximal error percentage of one percent and performs the simulation in less than a second.

2 Learning a default strategy

In the previous chapter we created a calculator that can determine the strength of a hand by calculating the probability of winning. In this chapter we shall use the calculator to calculate the strength of the hands.

When the APC first joins a poker game it has no information about the opponent, and in this case it must use a default strategy while it gathers more information.

In this chapter we will find a solution to the problem statement:

Problem statement 2

How can one develop a strategy without having any information about the opponents?

Even though players have different strategies they often have some decisions in common. Most players tend to play more aggressively the better their chances are of winning and likewise most players will fold if they have a weak hand. These tendencies can be used in a default strategy. The popular decisions are more likely to be good. For instance, most players agree that it is unwise to fold a pair of aces in the pre-flop.

The default strategy has to work against every strategy, therefore it is impossible for it to be better than all of them. The goal for the default strategy is not to win, although that is preferable, but instead to reduce the loses while it gathers information about the opponent.

2.1 Design

To develop a strategy in poker the two most commonly used options are to either directly program the procedures in the code or to create a self-learning algorithm.

Programming the procedures in the code requires the programmer to have a deep insight in how to play poker and how to make the optimal decisions during a game. One can also use the expertise of professional poker players in case one lacks the insight.

The self-learning algorithm uses the concept watch and learn by observing

other players and trying to learn the strategy behind their decisions. This method requires that the algorithm has someone to observe.

Since we do not have any particular insight in how to play poker and do not have expertise from any professional poker players, we will implement a self-learning algorithm. Additionally, by using this method the computer is not limited by our understanding of the game. We will use data from real life poker games for the algorithm. The data is further described in section 2.2.

To implement the self-learning algorithm we use an artificial neural network (ANN), see section 2.1.1. The ANN is well suited for finding patterns of the players decisions.

Our goal is to design an ANN with a total network error (TNE) of five percent or less. We find five percent to be acceptable as players often take irrational decisions and the ANN only tries to find an approximation of the results.

We use an iterative development method to design the ANN. We start by designing a simple ANN and then move on to more complex ANN's.

All our ANN's have exactly two outputs. The first output is whether or not to be defensive by checking or calling and the second output is whether or not to be aggressive by betting or raising. The closer an output is to the value one, the more certain the ANN is, that it is the correct decision.

All our ANN's use normalised inputs and a sigmoid function as transfer function. The reason we settle for a sigmoid function rather than a step function is because the sigmoid function allows us to see how certain the ANN is of each decision being correct.

2.1.1 Artificial neural network (ANN)

An artificial neural network (ANN) is inspired by the human brain. It can be used for pattern recognition or classification among other things. An ANN can take any number of inputs and return any number of outputs.

An ANN is made up of neurons that are connected into a network. Each neuron takes a set of inputs and returns a single output. The output of a neuron is sent to all the connected neurons. Each input has a weight that determines influence of the input. The neuron uses an input function to calculate the net input, usually the sum of all weighted inputs, and pass

it on to the transfer function. The type of transfer function determines the output. A step function returns zero or one if the net input is above a certain threshold. This is useful for logical functions. If one needs a value between zero and one a sigmoidal function can be used instead.

Figure 4 models a neuron. Here the weighted inputs w_{i1}, w_{i2}, and w_{i3} are all sent to the input function $\sum$ which then calculates the net input net_i. The transfer function f then calculates the output from the net input.

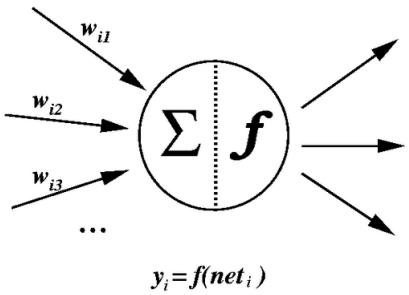

Figure 4: Model of a neuron.[10]

The neurons in an ANN are distributed in layers as shown in figure 5 and 6. The coloured circles represents neurons and the arrows represents the connections between the neurons. There are three layers, an input layer, a hidden layer, and an output layer. An ANN consists of one input layer and one output layer but may contain any number of hidden layers. The simplest type of ANN is the perceptron which has no hidden layers, see figure 5. It is used for single calculations. A multilayer perceptron is another type of ANN which contains hidden layers. It is used for more complex domains with multiple layers of computations.

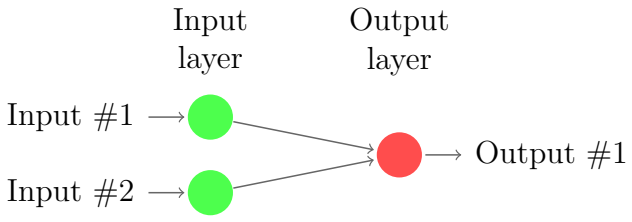

Figure 5: Perceptron with two inputs and one output.

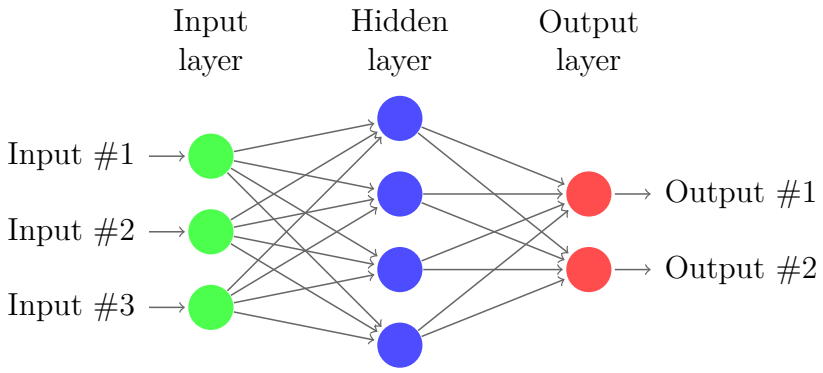

Figure 6: Multilayer perceptron with three inputs and two outputs.

The ANN can be trained using a training set of inputs. Using supervised learning, in contrast to unsupervised learning, one must also supply an expected output. For each input it will adjust the weights in order to get closer to the expected output. The TNE indicates the amount of training data that did not produce the expected result. The TNE is calculated during each iteration and the ANN will continue adjusting the weights until the TNE is acceptable.

Backpropagation is the most common algorithm for supervised ANNs. It adjust the weights from the end (the output neurons) back to the start (the input neurons).

After the training the ANN can be validated to see if it works. This is done using a test set different from the training set and see if the results of the ANN matches the expected results of the test set.

2.1.2 First ANN design

For the first attempt we design a simple perceptron that takes two inputs, and returns two outputs, see figure 7.

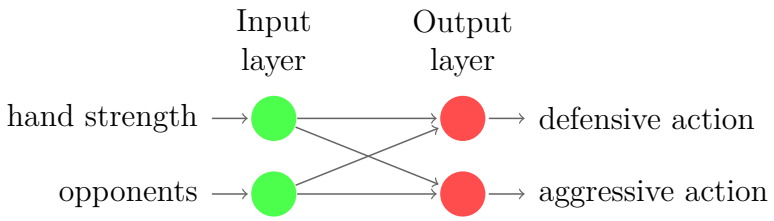

Figure 7: First ANN design.

The first input is the hand strength. We use the calculator from chapter 1 to calculate the probability of winning against a single opponent. The reason we always find the probability against a single opponent is because the probability of winning decreases drastically as the number of opponents increases. We use an absolute hand strength rather than a hand strength relative to the number of players. This makes it easier to compare the hands strengths in situations with different numbers of players.

The second input is the number of opponents. The input is normalized as

$$I_{norm} = \frac{Opp}{Opp_{max}}$$

Here Opp is the number of opponents and Opp_{max} is the maximum number of opponents (in our case nine).

From the tests we found that this network had a TNE of ∼19,9 %, see section 2.2.1. This does not fulfil our requirement of a TNE of five percent or less.

In order for a perceptron to be accurate the data have to be linear separable. This means that the outcomes can be separated by a single line if the are plotted in a graph. We plot some of the data in figure 8 to see if it is linear separable. The figure clearly shows that the data is not linear separable and therefore a single perceptron will not work.

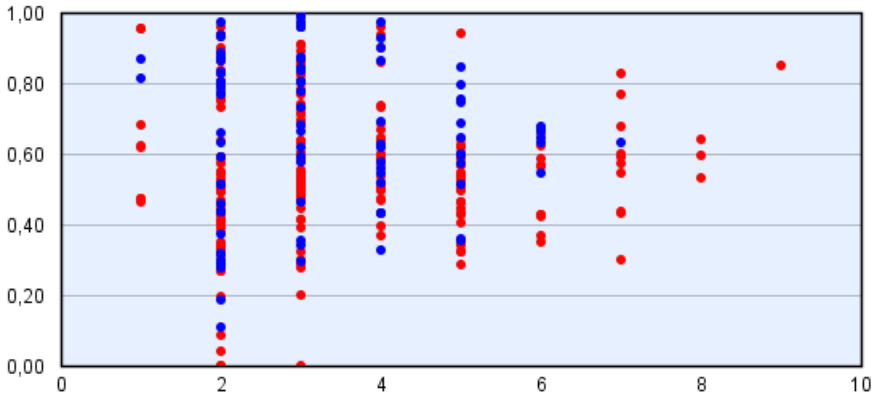

Figure 8: Distribution of actions. The x-axis the number of opponents and the y-axis is the hand strength. Aggressive actions are indicated by red dots and defensive actions by blue dots.

2.1.3 Second ANN design

Since the first approach designing a perceptron did not fulfil our requirement we instead design a multilayer perceptron (MLP) with five inputs, see figure 9.

The MLP is taking the same inputs as the perceptron from section 2.1.2, but now it takes three additional inputs: The chips of the player, the cost for the player to call, and the pot. We normalize the three new inputs as:

$$I_{norm} = \frac{I}{chips_{total}}$$

. Here I is the input to be normalized and $chips_{total}$ is the total amount of chips in the game, including the pot and the bets.

The MLP has one hidden layer with two hidden neurons. One hidden neuron to calculate the likelihood of winning and another for the economically aspect.

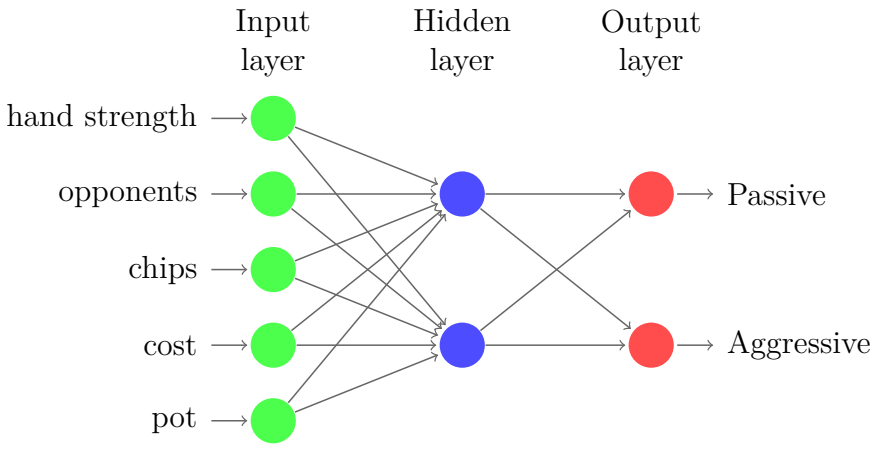

Figure 9: Second ANN design.

From the test in section 2.2.2, we can see that the TNE is ∼19,1 %. This is a small improvement compared to the first ANN but it still does not fulfil the requirement about a TNE of five percent or less.

2.1.4 Third ANN design

For our first ANN we use the same structure as the second ANN, see figure 9. We believe that the reason the second ANN did not work is because the data is found of random players. Each player may have a different strategy resulting in different actions. This will make it hard for an ANN to find a pattern.

Instead we use data of a single player in all games the player is relevant. The test in section 2.2.3 shows that the TNE varies quite a bit between the players. The higher the TNE are the harder the player is to predict. Since poker is a game of deception most players will be hard for our ANN to learn.

To further optimise the ANN, we try to adjust the number of hidden neurons, see section **??**.

2.2 Test

The University of Alberta has a research group that specializes in the field of artificial intelligence in poker [9]. They have released a dataset containing data from ∼18.000 real-life rounds of Texas hold'em limit poker. The dataset

only contains data about the hole cards of the players who made it to the showdown. We will refer to these players as relevant players.

The dataset consists of multiple files. The file called *hroster* contains the names of all players at the beginning of the round. The file called *hdb* contains the information about the community cards and the flop after each game state. For each player a file called *pdb.*<playername> exists. This file includes the hole cards (if the player did not fold), chips, profit from round and actions during each game state.

We have collected all the data from the different files into an single data file called *refactored-data.txt* to make it easier to analyse the rounds. We discarded the data for all rounds that has no relevant players. This is because no information about the hole cards exists in such rounds and therefore it is irrelevant for us.

refactored-data.txt will be used for training and testing the ANN's throughout this section.

For each test we create a new dataset. This dataset contains the information about the game for each action performed by a relevant player. All informations about the game is found in the moment of the action. For instance, an action in the pre-flop will have no information about the community cards. Each dataset contains data from 200 random poker rounds, resulting in ∼1200 actions distributed across all game states.

To create and test the ANN's we use a framework called Neuroph [11]. Neuroph is a neural network framework programmed in java.

2.2.1 Find the TNE of the first ANN

For this test the dataset only includes the normalized data about the hole cards of the player, the community cards and the number of opponents.

To find the TNE of the perceptron described in section 2.1.2, we train it using the dataset. The graph seen in figure 10 displays the TNE through each iteration of the training. From the graph we can see, that the TNE does not get any smaller than ∼19,9 %.

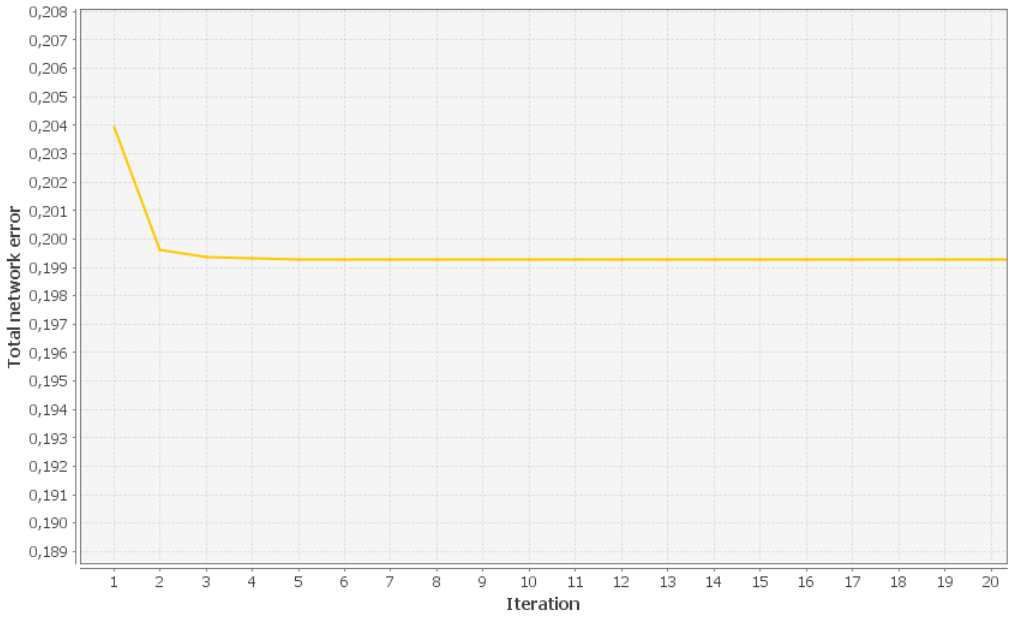

Figure 10: TNE graph from training the first ANN.

2.2.2 Find the TNE of the second ANN

For this test the dataset also includes the normalised data about chips, cost and pot.

We train the MLP from 2.1.3 and get the NTE graph shown in figure 11. The TNE does not get any lower than ∼19,1 %.

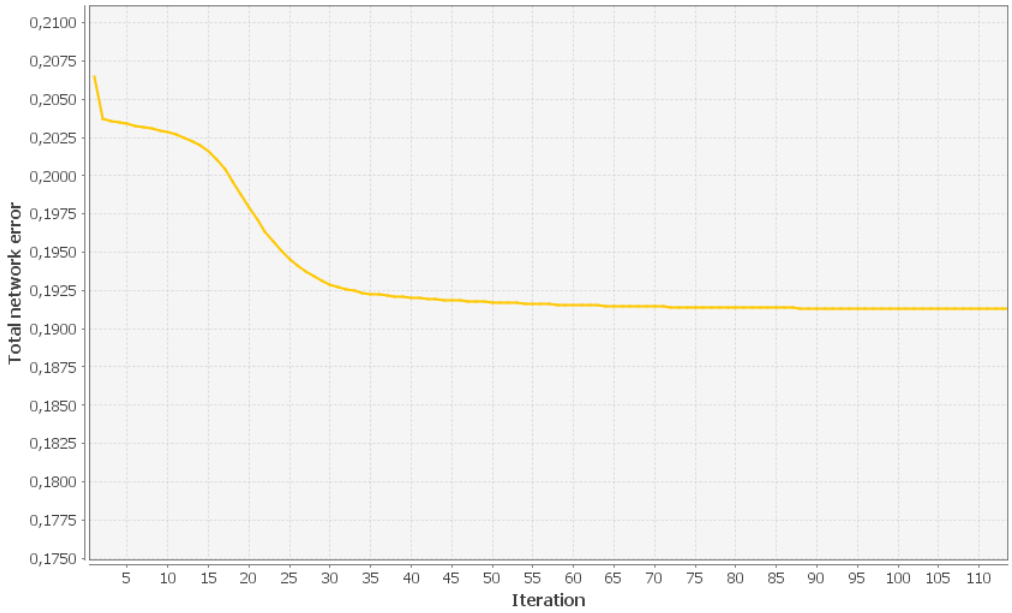

Figure 11: TNE graph from training the second ANN.

2.2.3 Find the TNE of the third ANN

For this test the dataset contains the same data as the previous test but all data is found for a single player. We try to find the TNE for three different players to see if this makes the ANN more accurate. The players we test are JimR, mayor, and PokiSimA.

For each player we train the MLP from 2.1.4 and find the TNE. The results are shown in table 7.

player	TNE (%)
JimR	∼18,7
mayor	∼14,5
PokiSimA	∼21,1

Table 7: TNE from training the ANN with different player data.

2.2.4 Finding the optimal number of hidden neurons

In this test we use the data from the same three player as in section 2.2.3. We construct three new ANN's with three, five, and seven hidden neurons. The TNE's can be seen in table 8. We have included the results from section 2.2.3 for comparison.

The number of hidden neurons does not seem to make the overall TNE get any lower.

#hidden neurons	JimR's TNE (%)	mayor's TNE (%)	PokiSimA's TNE (%)
2	∼19,4	∼14,5	∼21,1
3	∼19,5	∼14,0	∼20,9
5	∼19,1	∼15,0	∼21,1
7	∼17,8	∼14,9	∼21,3

Table 8: TNE from training ANN's with different numbers of hidden neurons.

2.3 Discussion

In this chapter we try to develop a default strategy by implementing a self-learning algorithm. We use ANN's to observe actions from other players and learn their strategy. Alternatively, we could have programmed the procedures of determining the action directly in the code. We ran into a few problems by trying to implementing the ANN.

The first problem is related to the data we use. The problem is that the data is not complete since we do not have the data about the hole cards of the irrelevant players. This means that we can not learn the ANN when to fold, as we have no input for the hole cards. To solve this we either need a complete data set or to program the procedure for determining when to fold.

Additionally, since we get the results as an action (defensive or aggressive), The expected results of each output is either zero or one, due to the

limitation of possible actions. These are the two extremes and it can be hard for the ANN to find an approximation.

The second problem was to determine the number of inputs for the ANN. We did not have much luck using only five inputs for our ANN. Since most players adapt their strategy to strategies of the opponents, it is close to impossible to learn the strategy of the player without knowing the strategy of the opponents.

The final problem we encountered was defining our requirements for the ANN. We do know what TNE is realistic and our requirements may be unrealistic.

It was a bad idea to try learn a default strategy from random player in random rounds. The players had most likely already gained information about the opponents and thus adapted their strategy to the opponents. It would have made more sense to observe players who just joined a table to see how they played in a situation where they had not adapted to the opponents yet. Additionally, trying to learn an approximation of every strategy is a bad idea. Observing bad players will cause the default strategy to take bad decisions. It would make more sense to observe only a few successful players. This way we can ensure the decisions to be smart.

2.4 Conclusion

In this chapter we try to answer problem statement 2:

Problem statement 2

How can one develop a strategy without having any information about the opponents?

We develop the strategy by creating an ANN and train it using data from real-life poker games. This solution did not work for us.

We trained our ANN to the strategy of random players by observing their actions and the information about the game in the moment of the action. We then found the TNE of the ANN's, and compared them to our requirement of having a TNE of five percent or less. Our best ANN had an TNE of $\sim$14.0 %

and therefore did not fulfil our requirement of five percent or less.

3 Learning to adapt to opponents

In the previous chapter we created an algorithm to develop the default strategy for the APC. The default strategy is one of the two strategies that will be used to determine the actions of the APC. The other strategy is the adaptive strategy.

The goal of this chapter is to learn the APC to create an algorithm to develop the adaptive strategy by solving problem statement 3.

Problem statement 3

How can one develop a strategy adapted to the opponents?

In poker there is no such thing as a single optimal strategy. Every strategy has weaknesses and therefore the optimal strategy is one that takes advantage of the weaknesses of the strategies of the opponents. In order to take advantage of the opponents strategies one must first understand their strategy. In poker understanding the opponents is one of the most important key elements of the game.

Once one understand the strategy of the opponent one has to adapt oneself's strategy.

3.1 Design

In order to learn the strategy of the opponents we try to model the player using player modeling, see section 3.1.1.

In this section we will start by introducing the concept of player modeling. We then design the player model we wish to use for the APC and finally implement it as a subsystem called the poker player model.

3.1.1 Player modeling

Player modeling is a loosely defined concept and may vary from one context to another. The concept of player modeling is to make a computational model of a player. This model includes game related attributes, such as play style and preferences, as well as non-game related attributes, such as

cultural background, gender, and personality. All decisions of the player are ultimately made on the basis of these attributes.

Player modeling is used to describe or predict the players decisions, reasoning and reactions. In the field of artificial intelligences the human player is the most used model for developing computer players. Understanding the reason behind every choice of a player will not only bring a better understanding of the player but also a better understanding of the game and its mechanics.

Since the player model can easily become extremely extensive one normally only includes the relevant attributes of the player.

3.1.2 Design of the player model

It is crucial to figure out which attributes are relevant to the specific player model. When trying to model a player there is almost no limit to what could be included. Attributes such as state of mind, energy, and distractions affect the decision of every human player.

We listed the attributes that we find most relevant for poker.

Aggressiveness How often does the player tend to bet or raise.

Tightness How strictly does the player's actions reflect the strength of the hand. For instance, a tight player will play aggressive when having a strong hand and defensive or fold when having a weak hand. A loose player may bluff (play aggressive having a weak hand) and slow play (play defensive having a strong hand) a lot.

Riskiness How easy is it to push the player to fold. Risky players tend to fold less often and are therefore harder to bluff.

Body language Most human players unconsciously show emotions through their body language. The professional poker players can tell a lot about a players hole cards solely by looking at their body language.

Time of decision making The time a player use for each decision can show the confidence of the players choice. A fast decisions indicates an easy decision.

Since our APC is targeted towards computer players as well as human players, it makes no sense to use attributes such as body language and the

time of decision making to model the opponents. Those attributes only affect human players. Instead we will model the opponents using the attributes aggressiveness, tightness, and riskiness.

Aggressiveness is easy to find simply by looking at the actions of the opponent. We choose to divide aggressiveness into three attributes. The first is overall aggressiveness and indicates the average aggressiveness of the player throughout the whole game. The second is recent aggressiveness which shows the aggressiveness within the recent five rounds. finally we have current aggressiveness which refers to the aggressiveness of the previous states of the current round. The reason we divide the aggressiveness into three attributes is to allow the system to register if the opponent changes strategy during the game. In such case the recent aggressiveness will differ significantly form the overall aggressiveness.

Tightness is a bit more complicated as we need to know the hole cards of the player. We only track the tightness of rounds where the player make it to the showdown. We will then find the average tightness for all rounds.

Riskiness is relevant because it shows how beneficial it can be to play aggressive. Safe players is easier to force to fold by simply playing a bit more aggressive than usual by for instance bluffing. Likewise it is a lot more risky to bluff against a risky player.

3.1.3 Implementation of the player model?

To model each player we have created a subsystem called the poker player model (PPM). The PPM is responsible for tracking the aggressiveness, tightness, and riskiness.

A PPM object is created for each opponent and the PPM then tracks the attributes of the player if given the data from a round.

The aggressiveness is found as:

$$Aggressiveness = \frac{A_{agg}}{A_{total}}$$

Her A_{agg} is the number of aggressive actions (bet or raise) and A_{total} is the total number of actions. This is found for every round. Overall-, recent, and current aggressiveness is found as an average of all rounds, last five rounds and current round respectively.

3.2 Test

3.3 Discussion

3.4 Conclusion

4 Discussion

5 Conclusion

Glossary

Poker

Keywords

Dealer	The players takes turn being the dealer in the beginning of each round. The player to the left of the dealer is always the first to take action in the bidding round.
Blind	A fixed amount that the two players has to pay in the start of each round.
Hole cards	A pair of private cards that is dealt to each player.
Community cards	The five public cards that are dealt to the table and are shared between all players.
Bidding round	In the bidding round the players takes turn performing an action. A bidding round will occur in every game state.
Action	An action can be performed when it is the players turn to act during the bidding round.
Pot	The sum of all the bids that have been placed in the round. The winner of the round wins the pot.
Chips	The amount of money a player has.
Round	A round consists of 4 game states: pre-flop, flop, turn, and river. In the beginning of each round the blind are paid.
Pre-flop	The first game state. In this state the hole cards are dealt.
Flop	The second game state. In this state the first three community cards are dealt.
Turn	The third game state. In this state the fourth community card is dealt.
River	The final game state. In this state the fifth community card is dealt. After the bidding round a showdown will take place.
Showdown	The phase where all players who have not folded show their hole cards and the winner is determined.
Hand	The best combination of the players hole cards and the community cards. See table 1.

Player actions

Fold	The player gives up and the player is out until the start of the next round. This action is always possible.
Check	The player does not bet any chips. This action is only possible if no other player has placed a bet.
Bet	The player bets an amount equal to the blind. All opponents have to match his bid in order to stay in the game.
Call	The player place a bet that matches the highest bid of the opponents.
Raise	The player place bet higher than the bet of the opponents. All opponents have to match this bid in order to stay in the game. This action is only possible if another player has placed a bet.
All-in	The player bets all his chips. The player is still in the game but he will not be able to act any more. This action is only possible if you cannot afford to call, bet or raise.

References

[1] caniwin. (no date known) Texas Holdem Heads-Up Preflop Odds [Online]. Available: https://caniwin.com/texasholdem/preflop/heads-up.php

[2] inferisx. (no date known) Top 10 popular card games [Online]. Available: http://topyaps.com/top-10-popular-card-games

[3] Marlos C. Machado, Eduardo P. C. Fantini and Luiz Chaimowicz *Player Modeling: What is it? How to do it?*, SBC - Proceedings of SBGames 2011.

[4] Aaron Davidson, Darse Billings, Jonathan Schaeffer and Duane Szafron, *Improved Opponent Modeling in Poker*, Department of Computing Science, University of Alberta.

[5] Garrett Nicolai and Robert Hilderman, *Algorithms for evolving no-limit Texas Hold'em poker playing agents*, Department of Computer Science, University of Regina, Regina, and Dalhousie University, Halifax, Canada.

[6] Jochen Fröhlich, *Neural Net Components in an Object Oriented Class Structure*, ebook pp. 11-28

[7] Poker Listings, *http://www.pokerlistings.com/poker-rules-texas-holdem*

[8] Lily Hay Newman, *Using AI to Study Poker Is Really About Solving Some of the World's Biggest Problems*, Slat blog

[9] University of Alberta Computer Poker Research Group, *http://poker.cs.ualberta.ca/*

[10] Artificial neuron models, *http://www.willamette.edu/ gorr/classes/cs449/ann-overview.html*

[11] Java Neural Network Framework, *http://neuroph.sourceforge.net/*

[12] Brains Vs. Artificial Intelligence, *https://www.cs.cmu.edu/brains-vs-ai*

A
Glossary