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 and was supervised by 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 deviate from their strategy or completely change it, 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 with up to nine opponents. The APC is programmed in Java.

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

As this thesis concerns Texas hold'em limit poker, we shall describe the basic rules and game play below. For a more in-depth description of Texas hold'em and other variants of poker see [2].

Game play

Texas hold'em limit poker is played with a standard deck of 52 cards and 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 are visible for everybody are dealt as the round progresses. We will refer to the hand of a player as the combination of the community cards and the hole cards of the player.

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 are quite simple. Each player has to create the best possible hand choosing five of the seven available cards (hole cards and 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 are not described in this section as they are irrelevant for this thesis. In 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 pair 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, J, Q, K, A)
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 having a straight all in the same suit.
Royal Flush is the same as a straight flush but the highest card of the straight has 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: Rank 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* [5] 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 play-

ers, 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"* [3]

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

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

Problem statements

1. How can APC determine the strength of a poker hand in any game state?
2. How can we develop a default strategy for APC without having information about the playing style of the opponents?
3. How can we further develop the APC's strategy to be able to adapt to the playing style of the opponent?

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 ∼0,15 seconds on average.

In chapter 2 we 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 APC managed to learn the strategies of three different players with a total network error ranging from ∼14 % to ∼20 %, but it still have room for improvements.

In chapter 3 we reflect upon problem statement three. We come up with a theoretical solution by using player modelling and artificial neural networks. We design a player model for the APC having seven inputs: overall aggressiveness, recent aggressiveness, current aggressiveness, overall tightness, recent tightness, overall riskiness, and recent riskiness. We design an artificial neural network having 68 inputs, 62 which are related to the player model, that is responsible for determining the action of the APC. The artificial neural network takes the play style of the opponents into account when determining the action.

Contents

1 Determine the strength of a poker hand

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

Problem statement 1

How can APC determine the strength of a poker hand in any game state?

The strength of a hand reflects the probability of winning therefore 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 of winning.

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 us 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.

Algorithm 1 shows the pseudo-code for determining the result of a single simulation.

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 examined 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 [1]. 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 using this number of simulations.

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 formula:

$$err = P_O - P_T$$

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

We inserted 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. From this we can see that 50.000 simulations fulfils our first requirements.

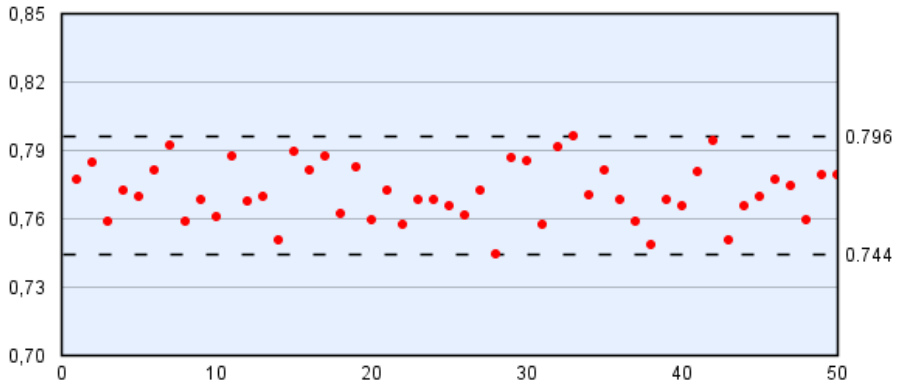


Figure 1: Result of the calculator with 1000 simulations

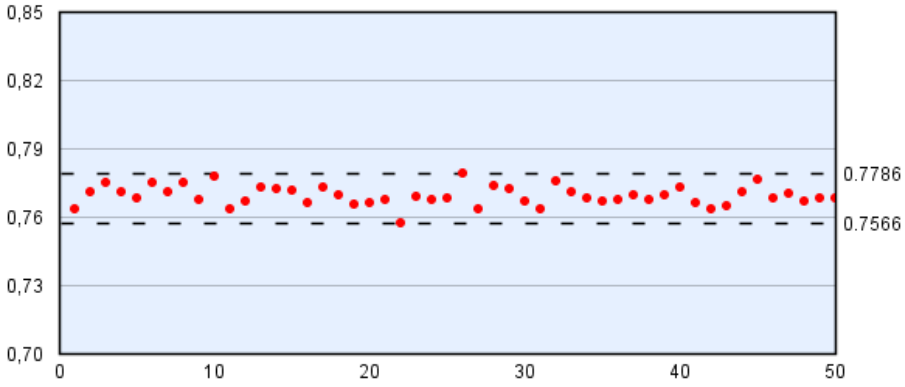


Figure 2: Result of the calculator with 10.000 simulations

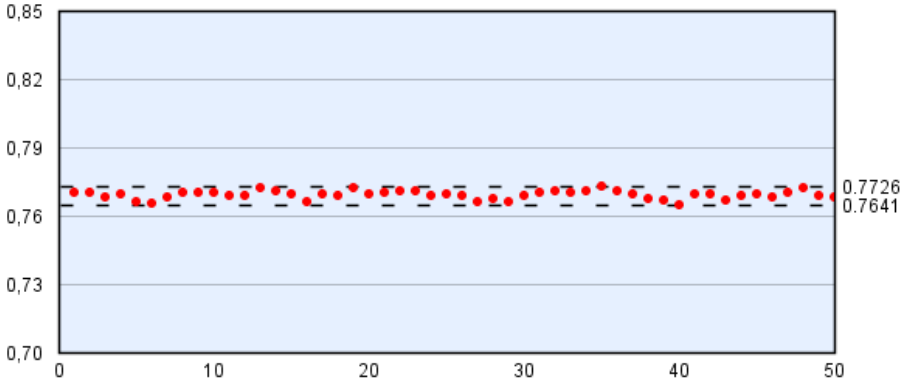


Figure 3: Result of the calculator with 50.000 simulations

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.

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 an 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 hands 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 percent 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 hands during all game states.

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 weak 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 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 exactly what was needed to fulfil 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.

Since we compare the results of the calculator to caniwin, we need to ensure the reliability of that source. Bobby, the author of caniwin, has a bachelor in math from the University of Southern California in LA. The rest of the thesis is written on the assumption that the results from caniwin is correct

Alternatively we could have created our own formula to calculate a rank, or used an existing one, for instance the Chen formula. However, this has been beyond the scope of this thesis.

1.4 Conclusion

In this chapter we have answered the question:

Problem statement 1

How can APC determine the strength of a poker hand in any game state?

We define the strength of a poker hand as the probability of winning with that hand.

We have implemented a subsystem called the calculator that can estimate the probability of winning with any given set of hole cards. The calculator uses the Monte Carlo method and it works for every poker state with up to ten players. 50.000 simulations met the requirements we made for the calculator.

The calculator has a maximal error percentage of one percent compared to the true probability and performs the calculation in ∼0,15 seconds.

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 this calculator in the process of developing a strategy for the APC.

When the APC first joins a poker game it has no information about the opponents, 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 we develop a default strategy for APC without having information about the playing style of the opponents?

Even though players have different strategies they often have some decisions in common. Most players tend to play more aggressively the stronger their hand are the bigger are their chances of winning and likewise most players will play defensive or 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 an algorithm that can learn a strategy from a player.

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 algorithm uses the concept watch and learn by observing other players and 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 a professional poker player, we will implement an 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 2.3.

To implement the algorithm we use an artificial neural network (ANN), see section 2.2. The ANN is well suited for finding patterns of the players decisions.

The total network error (TNE) is how great a deviation there is from the expected output to the actual output of the ANN. Our goal is to design an ANN with a 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, weighted sum as input function, 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.

The ANN's uses backpropagation for learning with a learning rule of 0,2.

2.2 Artificial neural network

An 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 as an input 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.

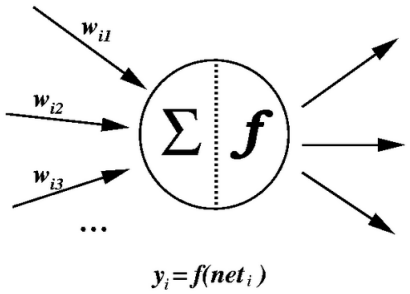


Figure 4: Model of a neuron.[11]

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. A multilayer perceptron is illustrated in figure 6.

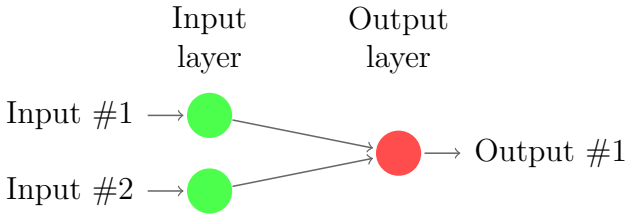


Figure 5: Perceptron with two inputs and one output.

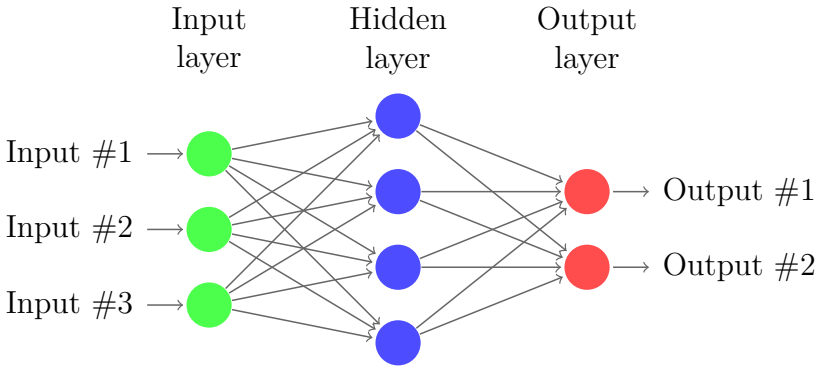


Figure 6: Multilayer perceptron with three inputs and two outputs.

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. During testing a TNE graph can be created to show how the TNE of the ANN decreases as the training progresses.

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

If the training is succesful and the TNE is acceptable one can validate the ANN using a validation set. The validation set should be different from the training set and it is used to see if the ANN has learned the entire domain of

which it has trained or if it has only learned the inputs given in the training set.

2.3 Test and traning the ANN

The University of Alberta has a research group that specializes in the field of artificial intelligence in poker [10]. They hxave 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. One file contains the names of all players at the beginning of each round. Another one contains the information about the community cards and the flop after each game state. For each player a file 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 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.

The new data file 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 [12]. Neuroph is a neural network framework programmed in java.

2.4 Our ANN's

Below we present the three ANN's that we have constructed as well as the test results. Each ANN is more complicated than the preceding one.

2.4.1 ANN 1

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

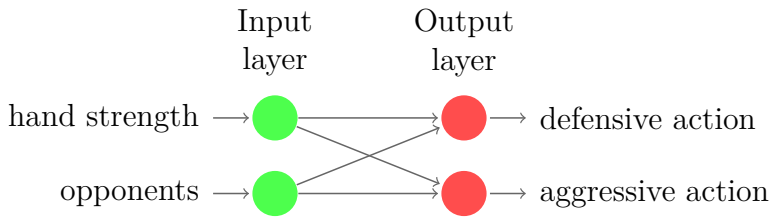


Figure 7: First ANN design.

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. In the start of a round all players are still in the game but it is unlikely that every player will make it to the showdown.

To avoid calculating a hand strength that will most likely become outdated as players fold, we instead find an absolute hand strength that is not affected by the number of players. Instead we give the number of players to the ANN and let the ANN find the pattern between the number of players and the actions.

We define an absolute hand strength as the hand strength versus one opponent. Using an absolute hand strength makes it easier to compare the hands strengths in situations with different numbers of players.

Since the inputs in the ANN has to be between zero and one we will have to normalize some of our input data.

The first input is the hand strength found using the calculator. It is already between zero and one so no normalisation is needed.

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

$$I_2 = \frac{Opp}{Opp_{max}}\,.$$

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

2.4.2 Test of ANN 1

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 2.4.1, we train it using the dataset. The graph seen in figure 8 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 %.

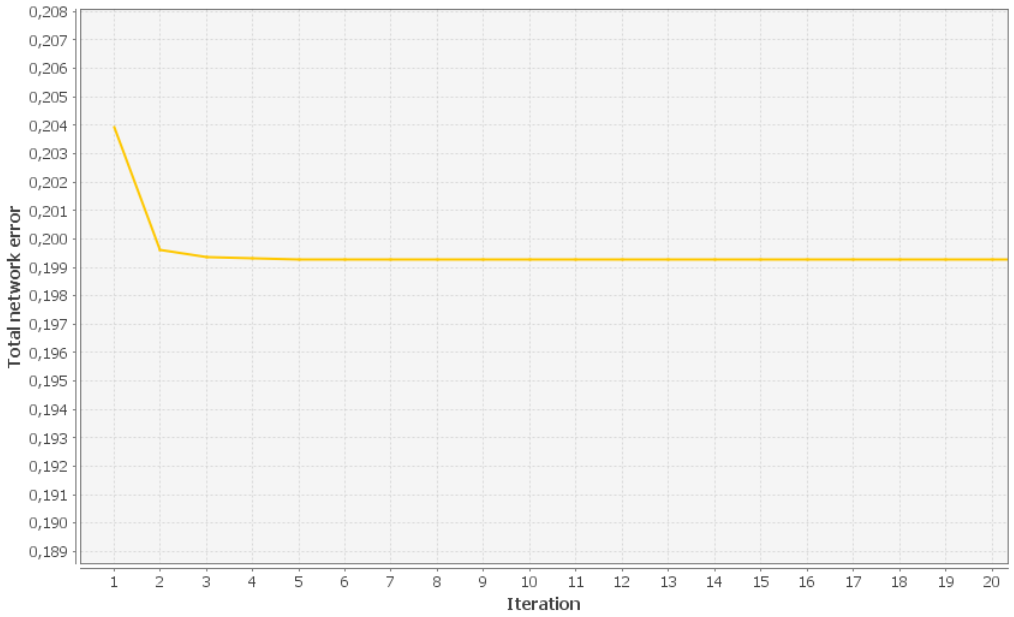
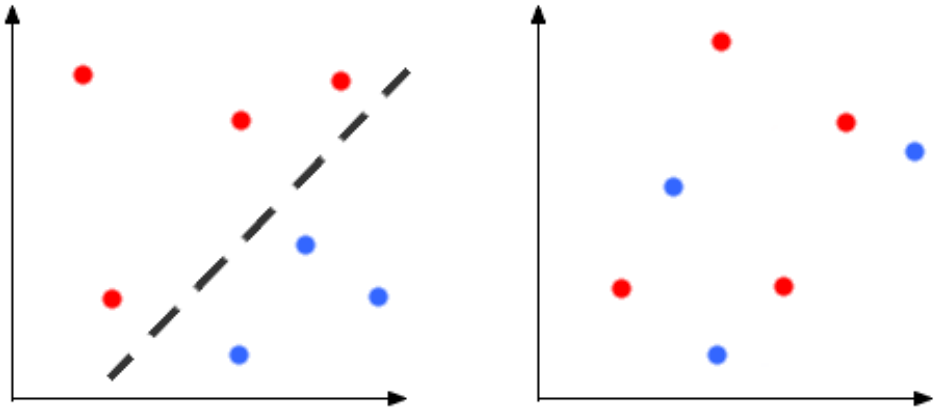


Figure 8: TNE graph from training the first ANN.

In order for a perceptron to be accurate the data have to be linear separable as shown in figure 9. This means that the outcomes can be separated by a single straight line if they are plotted in a graph.

We plot some of the data in figure 10 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.



(a) Linear separable data. (b) Non linear separable data.

Figure 9: Example of linear separable data.

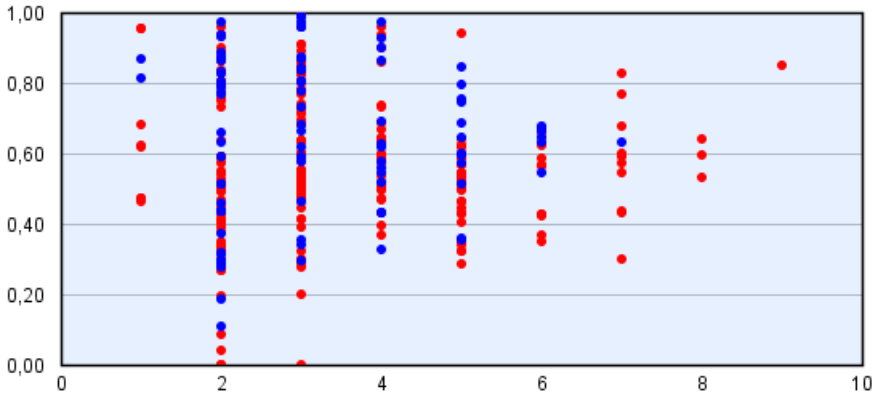


Figure 10: Distribution of actions. The x-axis the number of opponents and the y-axis is the hand strength. Aggressive actions are indicated by red dots and defensive actions by blue dots.

2.4.3 ANN 2

Since we would like the TNE to be equal to or less than five. We instead design a multilayer perceptron (MLP) with five inputs, in the hopes or lowering the TNE, see figure 15.

The MLP is taking the same inputs as the perceptron from section 2.4.1, and three additional inputs: The chips of the player, the cost for the player

to call, and the pot. Furthermore this ANN has two hidden neurons. We normalize the three new inputs as:

$$I_3 = \frac{chips}{chips_{total}}$$

$$I_4 = \frac{cost}{chips_{total}}$$

$$I_5 = \frac{profit}{chips_{total}}$$

Here *chips* is the APC's chips, *cost* is how many chips it costs to call, $profit$ is the amount of chips at stake and $chips_{total}$ is the total amount of chips in the game, including the pot and the bets.

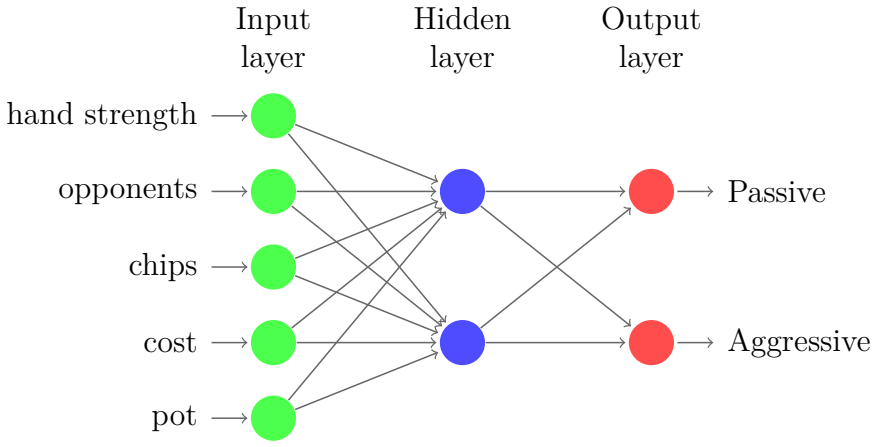


Figure 11: Second ANN design.

2.4.4 Test of ANN 2

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

We train the MLP and get the TNE graph shown in figure 12.

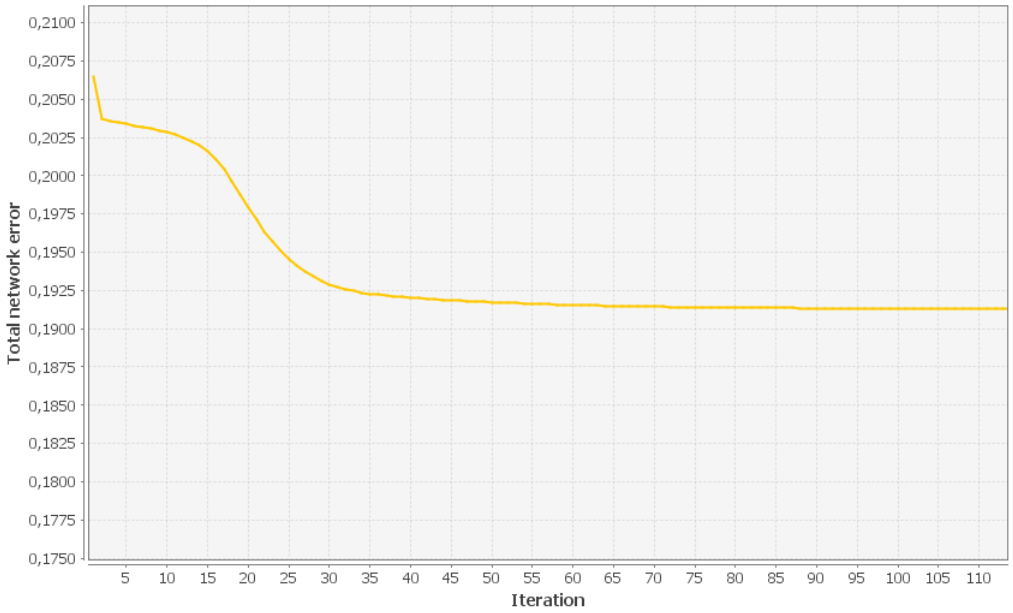


Figure 12: TNE graph from training the second ANN.

From the test in this section, we can see that the TNE does not get lower than ∼19,1 %.

We believe that the reason the second ANN did not work is because the data is found from 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.

There is a small improvement compared to the first ANN in the terms of TNE. It might be a small improvement but perhaps this is the right direction that we are heading.

2.4.5 ANN 3

For our third ANN we use the same structure as the second ANN, see figure 15. Instead of collecting data from random players, we use data of a single player in all games where the player is relevant. This hopefully result in the ANN being able to recognize a pattern in the players strategy.

2.4.6 Test of ANN 3

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 2.4.3 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.

There is a rule of thumb when determine the number of hidden neurons that the number of hidden neurons should be between the number of inputs and the number of outputs, Therefore to further test the ANN we try to test it using different numbers of hidden neurons within that range. We construct three new ANN's with two, three, four, and five hidden neurons. The TNE's can be seen in table 8.

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
4	∼19,6	∼14,2	∼21,0
5	∼19,1	∼15,0	∼21,1

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

The tests in this section shows that the TNE varies quite a bit between the players. The higher the TNE are the harder the player is to predict. Because poker is a game of deception, a player will often try to misguide the opponents by making illogical choices. This makes it hard for the ANN to figure out the players strategy. From this we can tell that mayor is probably not shifting his strategy as much as the two other players. Therefore mayor is the most preferable player when it comes to the success of adapting to another players strategy.

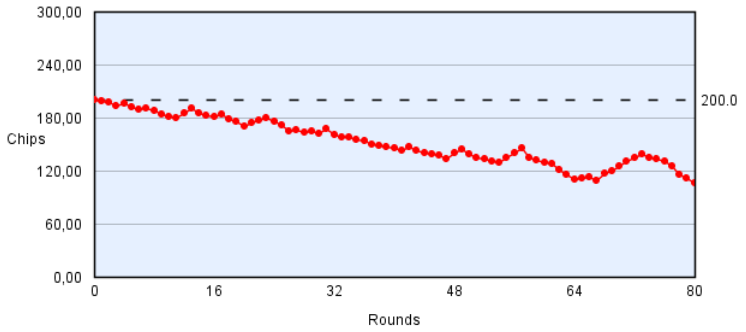


Figure 13: Graph of the profits change throughout the game

To further test whether or not the ANN can make profitable choices based on the training with the data from the player mayor, we choose to use a test-bed for poker bots called Poker Genius. Here we are able to test the ANN against the poker bots of Poker Genius. We manually recreated the choices the ANN made against the poker bots of Poker Genius. It was done manually, because we did not manage to make APC work for Poker Genius. As shown in figure 13 it is clear that the ANN is not able to compete on the same level as the poker bots from Poker Genius.

2.5 Discussion

In this chapter we try to develop a default strategy by implementing a self-learning algorithm. We use ANN's to observe the actions of other in order to learn their strategy. 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 players who fold. This means that we can not learn the APC when to fold, as we have no input for the hole cards.

If we want to observe a player and learn the strategy behind the players actions, we need to know the hole cards of the player. To solve this we need to find a new dataset with no hidden information about the player(s) we want the APC observe.

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. Zero or one are the two extremes and therefore it can be hard for the ANN to find an approximation.

Another problem we encountered was defining our requirements for the ANN. We do not know what TNE is realistic for an ANN for poker and our requirement of a TNE of five percent or less may be unrealistic. It would have been better set the requirement in regards to performance against other poker bots. For instance, a requirement could have been that the APC should should have more chips than the test bot after playing a game of 200 rounds.

Learning a default strategy based on random players in random rounds was not a good option. However we were not able to retrieve any game data from professional poker players. An alternative method would be to choose the most successful player in the dataset, but the rises the issue that just because a players strategy is good against one strategy does not make it good against another. Another issue is that in order for a player to be successful the player has to be good at adapting to the opponents strategy. This would mean that the ANN would have a hard time learning the strategy of the player, as the player may change the strategy or may use a completely different strategy if given other opponents. Our data was tracked for a game where people can join and leave as they please. It is therefore very likely that the players we observed changed strategy during the game.

The ANN were not able to make profitable choices when playing against the poker bots of Poker Genius. We believe the main reason for the bad performance was the fact that the APC never learned how to fold. We believe that this method for developing a default strategy could have worked if we were to get data from poker games where at least one of the players had no hidden information.

2.6 Conclusion

In this chapter we try to answer problem statement 2:

Problem statement 2

How can we develop a default strategy for APC without having information about the opponents?

We try to develop the strategy by creating different ANN approaches and train them using data from real-life poker games.

The tests were partly successful, as they show that we in fact were able to train an ANN to recognise the strategy of the players. We created several TNE graphs that display how the ANN becomes smarter as the training progresses. Although we did not manage to design an ANN that could learn the strategy of a player well enough.

By redesigning the structure of the ANN we managed to get an absolute improvement of the TNE of ∼6 % which is a relative improvement of ∼30 %.

Our best ANN design takes five inputs (hand strength, number of opponents, chips, cost, and pot) and produce two outputs (whether to play defensive and whether to play aggressive).

We tested it using three randomly picked players. We got an TNE of ∼14 %, ∼19 %, and ∼21 % for each of the players respectively.

We also tested the APC against a simple poker bot, and the APC end up losing roughly half its amount of chips in 80 rounds.

3 Learning to adapt to opponents

In the previous chapter we tried to create an algorithm to develop the default strategy for the APC.

The goal of this chapter is to further develop the strategy of APC to also adapt to the strategies of the opponents. This chapter will be mainly theoretical since it is based on the strategy of the previous chapter, which did not work.

Problem statement 3

How can we further develop the APC's strategy to be able to adapt to the playing style of the opponent?

In poker there is no such thing as a single strategy that is better than any other 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 strategy, one must first understand their strategy. In poker understanding the opponents is one of the most important key elements of the game.

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

Since poker is a game of deception the opponents may change their strategy during the game. In this case the player needs to find out about this and adapt the strategy once again.

3.1 Design

Because the work of this strategy is based on the ANN from chapter 2, the focus of this chapter is on the theory rather than implementation and testing. The reason is, that the ANN from chapter 2 did not end up working properly, and therefore it would not be possible to tell if the solution did not work, due to the original ANN or due to the improvements created in this chapter.

Instead of implementing the improvements, we just suggest how we could have implemented them.

There is two commonly used approaches to develop a strategy that adapts to the strategy of the opponents.

The traditional approach is to use statistics to learn about the opponents. Such an approach uses probabilities based on the players actions to determine the strength of the players hand. Such probabilities may include but are not limited to, the percentage of hand the player folds during each game state, the percentage of times the player bluffs on the river, and the strength of the hand the player has when calling aggression. This is useful against a player who use the same strategy throughout the whole game and who's actions are deterministic.

The other approach is using player modeling, see section **??** and artificial neural networks (ANN), see section **??**. This approach has gained increasing popularity through the recent years. In this approach the ANN is responsible of learning the strategy of the player. This approach is good for learning the strategy of a player who performs non-deterministic actions. It can also adapt in case the player changes strategy.

We choose to us the second approach and create a player model and an ANN. We find it more interesting to use player modeling and ANN's as this approach seems to have a lot of potential within the field of artificial intelligence in poker.

Aaron Davidson is a master of science from the University of Alberta. In his thesis about opponent modeling in poker [**?**], he uses ANN's in combination with player modeling in order to develop the poker computer Poki. An ANN was designed to predict the action of an opponent. The ANN uses a total of 17 player specific inputs. Using this design, Poki was able to win against most online poker players and computers but was far from world class.

G. Nicolai and R. Hilderman also made use of ANN's and player modeling in their thesis [**?**]. They use a single ANN to determine the action of the computer. The model each player using overall aggressiveness and recent aggressiveness and give it as an input to the ANN. Although their computer agent was no way as advanced as Poki, it still showed potential.

Using ANN's in combination with player modeling looks like variable approach. In the rest of this chapter, we will discuss this approach and suggest a potential way of implementing it. We will start by introducing the concept of player modeling. We then design a player model suitable for the APC and finally design the ANN.

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 has to determine which attributes are relevant.

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 player's 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 while having a weak hand) and slow play (play defensive while having a strong hand) a lot.

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

Body language Most human players unconsciously show emotions through their body language. The professional poker players can often tell a lot

about an opponent's hole cards solely by looking at the opponent's 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 can be found by simply looking at the actions of the opponent.

Tightness is a bit more complicated as we need to know the hole cards of the player. Therefore we can only track the tightness of rounds where the player makes it to the showdown.

Riskiness is also easy to measure. Riskiness shows how likely the player is to call while under pressure from opponents. We can determine the riskiness of the player by tracking the number of times the player folds contra the number of times the player calls or raises.

The APC needs to be able to recognise if a player changes strategy during the game. To accomplish this we split each attribute into the two attributes: overall and recent. Overall covers every round throughout the game while recent only covers the last ten rounds.

The reason we choose ten games for the recent attribute, is because a player might get lucky and get several strong hole cards in a row. This may cause the player to play more aggressive than usual even though the player still uses the same strategy. This will result in a miss read by the player model. We found ten games to be high enough to avoid miss reads and also low enough to actually track changes in the players strategy.

Aggressiveness is also split into one additional attribute called current aggressiveness, which is the player's aggressiveness in previous game states of the current round.

We end up using the following attributes for our player model:

- Overall aggressiveness
- Recent aggressiveness

- Current aggressiveness
- Overall tightness
- Recent tightness
- Overall riskiness
- Recent riskiness

In the rest of this section we will describe one way we could have implemented the player model. Note that the player model has not been implemented but this is only a description of how we would have done it.

The aggressiveness of a round n can be found as:

$$Agg, n = \frac{A_{bet}, n + A_{raise}, n}{A_{total}, n}$$

A_{agg}, n is the player's number of bets A_{bet}, n and raises A_{raise}, n in round n and A_{total}, n is the player's total number of actions in round n.

Overall aggressiveness can be found as the average aggressiveness of all rounds, recent aggressiveness would be found as the average aggressiveness of the last ten rounds and finally the current aggressiveness would be found as the average aggressiveness of the previous game states of the current round.

The tightness of the player can be found by backtracking the actions of the player after the round is finished. This is of cause only possible for rounds that makes it to the showdown, since we otherwise do not know the hole cards of the player.

In order to determine if a player either bluffs or slow plays, we would have to establish some boundaries based on the hand strength.

We took a sample of 100 defensive actions and 100 aggressive actions, and plotted the data in figure 14. The boundary for bluffing (red line) shows that the players tends to play defensive below this line. Likewise the boundary for slow play (blue line) shows that the players tends to play aggressive above this line.

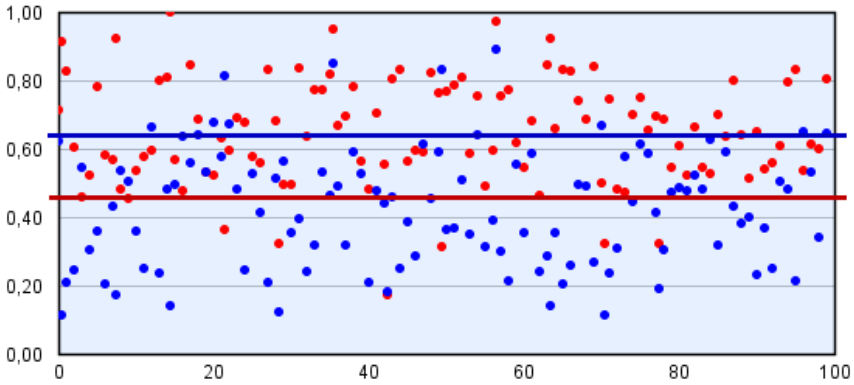


Figure 14: Distribution of actions in regards to the hand strength. Red dots indicates aggressive actions and blue dots defensive actions. The red line indicates the boundary for slow play and the blue line the boundary for bluffing.

We define slow playing as playing defensive with a hand strength of more than 0,63.

We define bluffing as playing aggressive with a hand strength of less than 0,46.

We can then find the tightness of a round n using the formula:

$$Ti, n = 1 - \frac{A_{bluff}, n + A_{slow}, n}{A_{total}, n}$$

Here A_{bluff}, n is the number of times the player has bluffed in round n, A_{slow}, n is the number of times the player has slow played in round n and A_{total}, n is the total number of actions in round n.

We can then find the overall- and recent tightness the same way as with the aggressiveness. Although we can only track the past ten rounds where the player made it to showdown.

Finally we can find the riskiness of a round n as the number of times the player calls or raise in regards to the number of times it costs the player to continue:

$$Ri, n = \frac{A_{call}, n + A_{raise}, n}{A_{call}, n + A_{raise}, n + A_{fold}, n}$$

Here we have the number of times the player calls A_{call}, n, raises A_{raise}, n and folds A_{fold}, n, all in round n.

3.2 Combining the ANN and player model

The ANN for the adaptive strategy would be based on the ANN design from section 2.4.3, see figure 15.

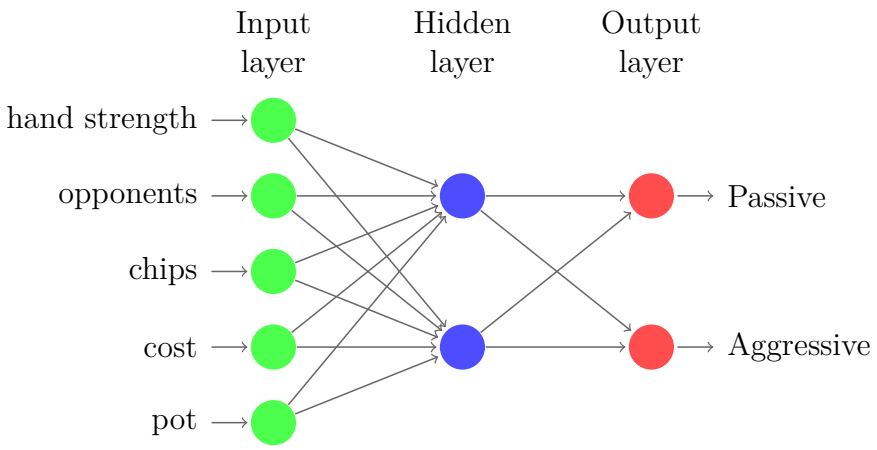


Figure 15: Second ANN design.

We choose to use the approach by G. Nicolai and R. Hilderman and construct a single ANN that takes the attributes of the players as inputs. The target is to learn the ANN what to do, based on the inputs about the game and attributes of the opponents.

The inputs for the ANN can be seen in table **??**. The attributes of the players are the attributes found in the previous section. In case there is less than nine opponents all attributes of a non-existing player is set to zero. Since the input function finds the net input as the sum of all the weighted inputs all inputs that are zero will not affect the ANN.

input	feature
1	hand strength of APC
2	number of opponents
3	chips of the APC
4	cost of the APC
5	pot
6 to 12	attributes of opponent 1
13 to 19	attributes of opponent 2
20 to 26	attributes of opponent 3
27 to 33	attributes of opponent 4
34 to 40	attributes of opponent 5
41 to 47	attributes of opponent 6
48 to 54	attributes of opponent 7
55 to 61	attributes of opponent 8
62 to 68	attributes of opponent 9

Table 9: Inputs for the ANN design

3.3 Discussion

In this chapter we discuss how we could have improved the strategy of the APC to also take the opponent's play style into account. Since we have no tests it is close to impossible to say if our solution would work or not.

We choose to use player modeling to model the opponents using seven attributes. The three main attributes we choose are aggressiveness, tightness, and riskiness. All these attributes are used to say something about the play style of the player. Other attributes more specific to the game state might also be worth considering such as: chips of the player, position of the player, the amount of chips the player has already committed in this round. Most of the inputs used for Poki are all related to the game state rather than the

play style.

We believe that it is a good idea to put all the attributes of the player model as inputs to the ANN. This way we can always try to remove some of the inputs or add new ones in order to measure if the TNE of ANN gets lower. This way it would be fairly easy to test which attributes that has an impact on the result.

Because of the huge amount of inputs the ANN would probability have to redesigned so it have more hidden neurons.
7

All together this approach seems very adaptable and easy to modify in order to incrementally test the ANN with attributes of different player models.

3.4 Conclusion

In this chapter we discuss and suggest a solution to problem statement 3.

Problem statement 3

How can we further develop the APC's strategy to be able to adapt to the playing style of the opponent?

One can either use a statistically approach or an approach using player modeling. We choose to use the approach using player modeling.

For our approach we use player modeling to find relevant attributes to define an opponent. These attributes will then be given as inputs to an ANN which then determines the action of the APC.

We find the following relevant attributes for the player model.

Aggressiveness refers to how often the player bets or raises.

Tightness refers to how strictly the player's actions reflect the strength of the player's hand.

Riskiness refers to how often the player pays to continue instead of folding.

To make it possible for the APC to detect changes in the strategy of an opponent we split each attribute into an overall attribute and recent

attribute. The overall attribute is the average of every round though the entire game whereas the recent attribute is the average of the last ten rounds. A change in strategy will affect the recent attribute and cause the APC to detect the change.

Aggressiveness has been split into one additional attribute called current aggressiveness which is the aggressiveness of the previous game states of the current round.

We end up with a total of 7 attributes:

- Overall aggressiveness
- Recent aggressiveness
- Current aggressiveness
- Overall tightness
- Recent tightness
- Overall riskiness
- Recent riskiness

We suggest the following way of finding the attributes.

The aggressiveness of a round can be found as the number of times the player bets or raise divided by the total number of actions of the player for that round.

The aggressiveness of a round can be found as the number of times the player bluffs or slow plays divided by the total number of actions of the player for that round.

The riskiness of a round can be found as the number of times the calls or raise divided by the total number of times the player has to pay in order to continue.

We then designed a new ANN that took the attributes for each a player as additionally inputs. The 68 inputs for the ANN can be seen in table 10.

input	feature
1	hand strength of APC
2	number of opponents
3	chips of the APC
4	cost of the APC
5	pot
6 to 12	attributes of opponent 1
13 to 19	attributes of opponent 2
20 to 26	attributes of opponent 3
27 to 33	attributes of opponent 4
34 to 40	attributes of opponent 5
41 to 47	attributes of opponent 6
48 to 54	attributes of opponent 7
55 to 61	attributes of opponent 8
62 to 68	attributes of opponent 9

Table 10: Inputs for the ANN design

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

[2] Poker Listings (no date known), *Hold'em Betting Rules: No-Limit, Limit, Pot-Limit* [Online]. Avilable:http://www.pokerlistings.com/texas-holdem-betting-rules

[3] Lily Hay Newman (2015, 01, 14), *Using AI to Study Poker Is Really About Solving Some of the World's Biggest Problems* line 46-52 [Online]. Available: http://slate.com/blogs/future_tense/2015/01/14/two_player_heads_up_limit_texas_hold_em_poker_weakly_solved_by_ai_researchers

[4] Inferisx (no date known), *Top 10 popular card games* [Online]. Available: http://topyaps.com/top-10-popular-card-games

[5] Carnegie Mellon University (2015, 04, 24 - 05, 08), *Brains Vs. Artificial Intelligence* [Online]. Available: https://www.cs.cmu.edu/brains-vs-ai

[6] University of Alberta Computer Poker Research Group (1995-2001), *Index of /IRCdata* [Online]. Available: http://poker.cs.ualberta.ca/IRCdata/

[7] Marlos C. Machado, Eduardo P. C. Fantini and Luiz Chaimowicz (2011), *Player Modeling: What is it? How to do it?* [Online]. Available: http://www.sbgames.org/sbgames2011/proceedings/sbgames/papers/tut/5-tutorial_player_modeling.pdf

[8] Aaron Davidson, Darse Billings, Jonathan Schaeffer and Duane Szafron (no date known), *Improved Opponent Modeling in Poker* [Online]. Available: http://poker.cs.ualberta.ca/publications/ICAI00.pdf

[9] Garrett Nicolai and Robert Hilderman (no date known), *Algorithms for evolving no-limit Texas Hold'em poker playing agents* [Online]. Available: http://www2.cs.uregina.ca/~hilder/refereed_conference_proceedings/icec10.pdf

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

[11] Genevieve Orr (1999), *Artificial neuron models* [Online]. Available: http://www.willamette.edu/~gorr/classes/cs449/ann-overview.html

[12] Neuroph, *Java Neural Network Framework* [Online.] Available: http://neuroph.sourceforge.net/

[13] Aaron Davidson, "Opponent Modeling in Poker: Learning and Acting in a Hostile and Uncertain Environment," M.S. thesis, Dept Computer Science., University of Alberta., 2002. http://poker.cs.ualberta.ca/publications/davidson.msc.pdf

A
Glossary