A MACHINE LEARNING APPROACH TO THE POKER PLAYING PROBLEM

Volkan Şirin, Aydın Göze Polat

Middle East Technical University
volkansirin@gmail.com
agplt@hotmail.com

Abstract
This article describes our approach to the poker playing problem. We used neural networks to teach the machine how people play poker. We worked on selecting the appropriate feature set and the neural network architecture for this specific problem. Resulting system has high prediction accuracy about playing style of an average poker player. This system can be used directly as a poker player bot or as the opponent modeling part of a more complex system.

1. Introduction
Poker is a game of imperfect information[1] in which players have only partial knowledge about the current state of the game. In Texas Hold’em poker game each player has 2 hidden cards that opponents cannot see until the end of the game. There are also community cards which can be seen and used by all players. These cards are put on the table one by one and at each step a betting round is conducted. So the game has stochastic outcomes since future cards to be opened to the table are not known. In addition to these challenges, in poker game folding (withdrawing) opponent does not have to show its hand which makes learning even more difficult. Finally, it’s a multi-agent game in which exploiting opponent’s weaknesses is essential for profit.

These properties bring many difficulties. Traditional game playing methods like deep search have not been sufficient and dealing with imperfect information is the main reason that progress on strong bridge and poker programs has lagged behind the advances in other games [2].

Different machine learning techniques are proposed to solve some aspects of the problem. First and foremost, ML methods are used for modeling opponents. Understanding the playing style of the opponent is very important in poker game. For example if our opponent is a loose player (i.e.: can easily get afraid and fold) bluffing would be the right course of action.

This game demands players to ‘co-evolve’ while playing. That means the bot should adapt its game-play, after a number of games with the same opponent. Alternatively a bot with a ‘standard’ game-play can be implemented but these kind of bots are both predictable and not very successful against a human player. Thus, creating an adapting bot that is generally effective against human players is a challenge.

Our idea was to use machine learning to learn average playing style of humans. We employed neural networks for this purpose. This module can be used in two ways. First it can directly provide a poker playing bot. Second, it can be used in a more complex poker playing system as an opponent modeler system. As a direct poker player bot, the results are promising: the average prediction accuracy is around 85%.

2. Literature Survey
University of Alberta Poker Research Group was pioneering the research in this particular area for the last decade. They have developed the poker bots called Poki, PSOPTI and VEXBOT [6]. Each of these bots used a different strategy.

Some of the major themes of their research have been the following:

- Knowledge-based Methods and Simulation are used to determine hand strength and potential which helps decision making [7]. Poki is based on these methods.

- Opponent Modeling methods to identify potential weaknesses [4].

- Game-Theoretic Methods are used to determine how rational agents should behave in multi-agent environments mathematically. Researchers are proposing computational methods to apply game theory-based approaches to large real world games of imperfect information, such as [8]. PSOPTI is also based on these methods [6].

- Game Tree Search is also used in poker context. Miximix search algorithm has been proposed to evaluate game trees in poker game [5].

Darse Billings’s PHD thesis is a good summary of their efforts. There are also important works outside of the University of Alberta.
R. Wahab uses a reinforcement learning agent with a high learning rate[18]. He tries his approach on 1-Card Poker which can be considered as a simpler version of Texas Hold’em. Considering extension to N-players, he concludes that the size of the state representation would not be a limiting factor. However, multiple opponents could slow the learning rate significantly. He says extending the approach to more complex poker variants increases the complexity of the state representation as well as slows the learning rate.

Developing a strategy by playing an opponent several times while ‘tuning’ some learning parameters is another similar approach proposed by Wildig and Kendall[19].

R. Booker, on the other hand, believes that leaving the raw assessment of the situation purely on some reinforcement learning agent would arise problems[20]. He proposes giving the agents some processed data and let them react accordingly. He also makes use of opponent modeling.

Among different types of Texas Hold’em, G. Carter analyzes the dynamics of underlying no-limit tournament poker-play, studying the non-academic poker experts' suggestions empirically. Moreover he introduces evolutionary algorithms as a method of searching the strategy space.[21]

Applying reinforcement learning to problems with a large state space, comes with various drawbacks such as:
1. A ‘modest’ state representation is necessary to limit the high memory cost.
2. As number of player N increases, complexity increases exponentially. (i.e. Let’s assume the complexity O(s) represents the memory requirement for 1 player. If N becomes 2, complexity becomes O(s^2) and required number of games to train also increases exponentially [18]. One can try to further simplify the state representation from the start or each time N increases, however this will affect correctness.)
If above problems are to be solved, then adaptation to various game styles is possible. This can be a big advantage when playing against a specific opponent.

3. Task Definition

Goal of a poker playing system is the finding of the most profitable action given a table state. Table state consists of previous actions of opponent and the player in the game. We mean maximizing the total income across a game sequence with a particular opponent when we say the most profitable action. So losing some money in a particular game can be the right course of action if it causes more income in the long run.

4. Neural Networks for Learning Average Play

Artificial Neural Networks are used to find complex relationships between a set of inputs and outputs. We used neural networks to understand the average playing style. With some additional features neural networks can also be used to understand our opponent’s playing strategy. In poker game it is usual that an opponent will play sub-optimally. Thus, a maximizing agent will out-perform an optimal agent against sub-optimal players [11].

In poker context, outputs of the network are obvious: fold, call and raise actions. We used various features describing the game state as inputs to neural network. Full list is below:

- Betting round
- Hand Strength (or Winning Odds for 1st phase)
- Negative Potential
- Positive Potential
- Player is Dealer or Not
- Player’s total money
- Opponent’s total money
- Player’s number of total raises
- Opponent’s number of total raises
- Player’s number of preflop raises
- Opponent’s number of preflop raises
- Player’s last action
- Opponent’s last action
- Player’s total bet
- Opponent’s total bet
- Total money in the pot

Instead of feeding card values directly to the neural network we use their strength and potential as features. This provides a reduction in input complexity since many hands (card combinations) has similar strength and potential values. We will introduce these measures in detail in next section.

Currently neural networks for preflop and river contain 14 input neurons, one hidden layer of 11 neurons and 3 output neurons. Neural networks for turn and flop contain 16 input neurons, one hidden layer of 11 neurons and 3 output neurons. Difference is that positive potential and negative potential is not used in first and last phase of the game. In last phase all board cards are opened so it is not meaningful to talk about the potential. In first phase of the game Winning Odds calculation also include potential information since it is computed for considering possible opponents cards and feature cards. Details are presented in next section of our design report.

5. Hand Evaluation Algorithms

The Hand Evaluation Algorithms aims to assess the strength of the hand of the agent. There are different
approaches to evaluate the hand in pre-flop phase and post-flop phase.

**Pre-Flop Hand Evaluation**

There are \( C(52,2) \) = 1326 possible hidden hand pairs in poker. The expected income rate of each hand is found using a simple technique that is called roll-out simulation. The simulation consists of playing several million trials. In each of these trials, after hidden cards are dealt, all other cards are dealt without any betting and the winning pair is determined. Each trial in which a pair win the game increases the value of that particular hand. Of course it is an oversimplification of the game, but it provides an accurate description of relative strengths of the hands at the beginning of the game.

David Sklansky, a professional poker player and author of two important books [7][9] about Poker, gives hand rankings for this phase of the game. His table is based on expert experience. There is a strong correlation between his rankings and the results of the roll-out simulation [2].

We used pre-calculated simulation results in our design. Some of the results are given below as an example:

- AAo........%85.14
- KJo........%82.22
- QJo........%79.63
- JJo........%77.2
- TJo........%74.57
- 99o........%71.08

**Post-Flop Hand Evaluation**

**Hand Strength.** The hand strength, HS, is the probability that a given hand is better than that of an active opponent.

All of the possible opponent hands are enumerated and checked whether our agent’s hand is better, tied or worse. Summing up all of the results and dividing the number of possible opponent hands gives the hand strength [2].

Below is the algorithm for hand strength calculation taken from [11]:

```c
HandStrength (ourcards,boardcards)
{
    ahead = tied = behind = 0
    ourrank = Rank(ourcards,boardcards)

    /*Consider all two card combinations of the remaining cards.*/
    for each case(oppcards)
    {
        opprank = Rank(oppcards,boardcards)
        if(ourrank>opprank)
            ahead += 1
        else if(ourrank == opprank)
            tied += 1
        else
            behind += 1
    }
    handstrength = (ahead + tied/2) / (ahead + tied + behind)
    return(handstrength)
}
```

**Hand Potential.** Hand potential calculations are for calculating the winning probability of a hand when all the board cards are dealt. The first time the board cards are dealt, there are 2 more cards to be revealed for each round. We calculate the potential impact of these cards. The positive potential, PPot, is the chance that a hand which is not currently the best improves to win at the showdown. The negative potential, NPot, is the chance that a currently leading hand ends up losing.

PPot and NPot are calculated by enumerating all the possible cards for the opponent like the enumeration in hand strength, in addition by enumerating all the possible board cards to be revealed. For all the combinations of possible opponent cards and possible board cards we calculate the PPot and NPot as:

PPot: Count the number of times our agent’s hand is behind, but ends up ahead
NPot: Count the number of times our agent’s hand is ahead, but ends up behind

Pseudo code and complete description of Hand Potential Algorithm can be found in [11].

We also used Winning Odds statistic, which is the winning probability given hands of the players. It is again calculated with simulations. Since we do not know the hidden cards of the opponent, we integrate over its probability distribution in order to calculate winning odds. Details of opponent hand probability distribution will be explained later.

6. Neural Network as a Poker Player

Given any table state our system is capable of returning an appropriate response action. First, we calculate hand strength and hand potential values using specified algorithms. After that, we extract other features from the table state and feed all of them to the neural network.

Three outputs of neural network can be considered as an indicator about strength of the corresponding action. At this point we can select the action which corresponds to the highest output node. Another way is to select randomly one of the actions weighted with the value of each output node. This approach strengthens the unpredictability of the system.
Embedding Neural Networks to a Game Tree

If we exchange features describing the state of the bot with the state of the opponent for the input, neural networks become an opponent modeler. Details of our neural network design are given at fourth section.

Game Tree Search in Poker

For the game tree of the limit poker, the branching factor for each node is three; each branch corresponds to one of the actions fold, call or raise. Figure 1 shows an example game tree for limit poker. Number of raises is limited to two for sake of simplicity. This game tree is constructed for the case our agent bets. Leaves of the tree shows expected values assuming our opponent has the winning hand and previously there is $35 in the pot.

The probabilities for branches leading to opponent nodes come from opponent modeling module. Darse Billings proposed Miximax algorithm for evaluation of game trees of imperfect information [5]. In this algorithm in order to determine the expected value of an agent node, values coming from each branch is averaged, each branch is weighted with assigned probability. So the calculation is:

$$EV(O) = \sum_{i \in \{f,c,r\}} Pr(O_i) \times EV(O_i)$$

So all possible opponent decisions are mixed and the agent is assumed to choose always the maximum EV branch. This is why it is called Miximax algorithm. If the agent uses a chance factor to determine its action than the algorithm is called Miximix algorithm [5].

Expected values at fold leaf nodes are straightforward. In order to determine expected values at call leaf nodes opponent probability distribution is used. Using this distribution winning probability ($P_{win}$) is calculated. And then:

$$EV_{Leaf} = P_{win} \times PotValue - (0.5 \times PotValue)$$

Half of the Pot Value is subtracted because it is the cost of reaching the leaf node.

While expanding the game tree, after each of the hypothetical opponent action, the probability distribution is updated and assigned to the following agent node.

Predicting Opponent’s Hand

We keep predictions about opponent’s hand in a probability distribution table. At the beginning of the game each possible hand has equal probability. As the game progresses and new actions of our opponent are observed, the probability distribution is updated accordingly.

Neural Network gives probability of a particular opponent action given its hand and the table state. So we can update the probability distributions using Bayes’ Law as follows:

$$POH, OA, TS) = \alpha \times P(OH, OA, TS) \times P(OA, TS)$$

OH denotes Opponent Hand, OA denotes Opponent Action and TS denotes Table State. $\alpha$ is the normalization coefficient.
7. Experimental Evaluation

Data Set

There are several hand history databases describing thousands of hands played before. For example, the database located at [15] is a very informative one. We downloaded 500 hand histories each describing a 1-1 limit Texas Hold’em Poker game, since we focused on this type. Unfortunately, these files are prepared for human inspection so we need to parse them in order to extract the features necessary for learning the game. Hence, there is no fixed attribute set. We can derive different feature sets from the same game. Our selections are listed in the fourth section.

Implementation and Methodology

The raw data set is consisted of 500 txt files each describing a full game played before. We developed a parser to acquire necessary attributes from these files. As mentioned before, we converted card information to hand strength and hand potential (negative and positive). To do this job we used Hand Evaluator library which can be downloaded from [16]. We developed a second parser to convert card information and other 12 features to real numbers.

Aforge ML library [17] was used when implementing four different feed forward neural networks. These four neural networks have been trained using the corresponding training data. Each neural network is specialized to guess the ‘next action’ of the player in specific phase of the game. ‘Next action’ can be one of these three: ’call/check’, ’raise/bet’ or ’fold’.

During training for fold actions outputs are set as (1, 0, 0), for call actions outputs are set as (0, 1, 0) and for raise actions outputs are set as (0, 0, 1).

In testing phase the output node which gives the maximum value is taken as the next action.

Since some phases of some games have more than one round, our final feature data set is consisted of: 592 data points for preflop phase, 764 data points for flop phase, 717 data points for turn phase and 649 data points for river phase. For each neural network last 100 data points are reserved for testing and others are used for training.

Results

For the actions ‘Call/Check’ and ‘Raise/Bet’ the results were promising. However, our current dataset does not contain enough number of games in which the action ‘Fold’ occurred. Only around 1-2% percent of the games contains fold and this is simply not enough. Because of this, the neural networks could not be able to guess the fold action at all.

Below is the evaluation of the results:

<table>
<thead>
<tr>
<th></th>
<th>Call Precision</th>
<th>Bet Precision</th>
<th>Call Recall</th>
<th>Bet Recall</th>
<th>Overall Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preflop</td>
<td>0.88</td>
<td>0.8</td>
<td>0.81</td>
<td>0.87</td>
<td>84%</td>
</tr>
<tr>
<td>Flop</td>
<td>0.73</td>
<td>0.88</td>
<td>0.92</td>
<td>0.63</td>
<td>78%</td>
</tr>
<tr>
<td>Turn</td>
<td>0.84</td>
<td>0.89</td>
<td>0.88</td>
<td>0.91</td>
<td>87%</td>
</tr>
<tr>
<td>River</td>
<td>0.95</td>
<td>0.89</td>
<td>0.83</td>
<td>0.97</td>
<td>91%</td>
</tr>
</tbody>
</table>

where

Precision = true positives / (true positives + false positives)
Recall = true positives / (true positives + false negatives)

Overall accuracy column shows the percent of accurate predictions made by neural network. This column also accounts for fold actions but since fold actions are also few in the test set, this hurts little.

Average prediction accuracy is %85.

While finding the correct neural network configuration we experimented with different number of nodes and layers. Below are the results:

Discussion

In Poker game there is no correct action. In the same situation one player can advise playing “bet” other may advise playing “call” so it is not possible to predict a player’s action 100 percent correct. Considering this we believe our results are good.

We processed raw input and created some meaningful training data. Giving the neural network processed data such as ‘Hand Strength', 'Negative & Positive Potential' and 'Opponent’s Number of Raises' has advantages, since it simplifies the state space without losing critical information and gives a priori information about opponent and the current state.

To achieve a certain level of performance (in our case, around 85% overall accuracy ), playing excessive number of games, as in the case for reinforcement learning, was not necessary. Training the network (in our case, a couple of minutes) was enough.

Using opponent modeling may fail, when the old opponent leaves and a new opponent comes, regardless of playing strength. For example after playing an experienced poker player, it is still likely that opponent modeling will fail against an amateur. However a neural network will achieve a generally good performance and it is robust against such changes. Yet it is hard to further improve the performance of a neural network.
8. Related Work
The most similar work to ours’ in the literature is of Aaron Davidson et al [4]. They also used neural networks to predict opponent’s next action. However they use a single neural network for all the stages of the game.

In our first design, we also used one neural network for all the three phases of the game after the flop. But this architecture has a drawback; it uses the information gained in a phase of game in another phase. However, players usually adopt different strategies for different stages of the game. So in our second design, we used four different neural networks for each phase of the game and we observed that prediction accuracy is improved.

Aaron Davidson reports 81% prediction accuracy in the average. Our prediction accuracy is 85%. This is probably because of our improved neural network architecture and careful feature selection.

9. Conclusion
The biggest difficulty we experienced was finding an appropriate data set. 500 data points may not be enough since poker game is very complex. But our results are surprisingly good. This is probably because of hand evaluation procedure. We believe that if size of the training set is increased, performance will also increase.

Because of limited data set, our system was not able to learn the fold action very well. Currently we defined a simple rule to overcome this: when the winning odds are below a certain threshold, the agent automatically folds.

Using neural networks directly as a player has one major drawback: it’s essentially a static player. So if the opponent understands the playing style of the bot he can exploit this. To overcome this, our current idea is to randomize the behavior a little.

Careful feature selection and a separate neural network for each poker phase, produced a generally good, fast and efficient system. Our work proves that neural networks can be a useful tool in artificial poker playing.

10. Future Work
Opponent modeling increases the performance when playing against a certain type of player for a long time. Although we defined 'looseness factor' and 'aggression factor' for this purpose, there can be a way to achieve opponent modeling directly in the neural network: Continue training the neural network while playing: -First, collect the opponent's moves (until now) and create similar artificial training data,

-Then further train the neural network with this specific data to 'learn' the opponent's 'style'.

-Size of the training data can be adjusted so that the neural network responses in reasonable time.

Evolutionary algorithms can produce agents that can adapt to the style of a specific opponent, and outperform a static competent player[19].

Above ideas can be applied to our work for further improvement.

References