CMPUT 499 - Research Project Review

Using Artifical Neural Networks to Model Opponents in Texas Hold'em

Aaron Davidson

email: davidson@cs.ualberta.ca

November 28th, 1999

Abstract:

This paper describes a system for predicting an opponent’s next action in the game of Texas Hold’em, a common Poker variant. Network performance in a variety of situations is shown, as well as compared to current opponent modeling systems. A technique for graphical representation of neural networks is also discussed as a useful tool for feature discovery.

Keywords: Neural Networks, Poker, Opponent Modeling, Feature Discovery

1. Introduction

The current state of opponent modeling in our poker project is very crude. Every game played against an opponent gives us a huge amount of information on how they play. Currently we remember a few action frequencies in a very limited context (bets-to-call and game stage). The context which a player may use to assist in making decisions is far richer than just those two factors. For instance, the number of players in the game, the size of the pot, draw potentials, and their position in the betting are all known to strongly affect the way a player will behave. By ignoring all of this context, we are discarding volumes of valuable opponent information. Different players will have different sensitivities to particular contexts. The problem we are faced with is sorting out what factors within all of this data actually affect a particular player's decisions.

At the heart of the opponent modeling problem there are two things that need to be discovered. First, there is the need to determine what hand a player is holding, based on their actions during the game. Second, we need to know what a player will likely do in any given game situation. For instance, it is valuable to know that if we re-raise a particular opponent on the turn when they appear to have a reasonably strong hand, that they will fold rather than call us down.

Artificial Neural Networks (ANN's) are well known for their ability to learn and identify patterns in noisy data. This paper will show ways in which ANN's can be used to enhance a poker playing program.

2. Architecture and Methodology

2.1. Training the Network

In order to predict an opponent's next action, we first need some observation data to train the network on. By logging game contexts and the associated observed actions from IRC poker games, training data was collected for a variety of different players. Context data is translated into an array of real numbers with range 0 to 1. **Fig. 1.** shows a table of the contextual information used. The numbers correspond to the input node in the network.

This data is then fed into a standard feed forward ANN (also known as a multilayer perceptron) with a four-node hidden layer and three output nodes. The sigmoid activation function was used for all nodes in the network. Each of the three output nodes represents the network's prediction that an opponent will fold, call, or raise

#	type	description
1	Real	Immediate Pot Odds
2	Real	Bet Ratio: bets/(bets+calls)
3	Real	Pot Ratio: amount_in / pot_size
4	Boolean	Committed in this Round
5	Boolean	Bets-To-Call == 0
6	Boolean	Bets-To-Call == 1
7	Boolean	Bets-To-Call >= 2
8	Boolean	Stage == FLOP
9	Boolean	Stage == TURN
10	Boolean	Stage == RIVER
11	Boolean	Last-Bets-To-Call > 0
12	Boolean	Last-Action == BET/RAISE
13	Real	(#players Dealt-In) / 10
14	Real	(# Active Players) / 10
15	Real	(# Unacted Players) /10
16	Boolean	Flush Possible
17	Boolean	Ace on Board
18	Boolean	King on Board
19	Real	(#AKQ on Board) / (# Board Cards)

Fig.1. Context Information used to train the networks. Boolean values are input as a 0 or 1, Real values as a real number from 0 to 1.

respectively. An output node can give a real value from 0 to 1, so by normalizing the output nodes, we are given a ready to use probability distribution (known in our existing poker system as the "Probability Triple" data structure). By training the network on all of our data, and using back-propagation to perform a gradient descent on the connection weights in the network, the network begins to successfully discover the importance of each input feature with regards to the opponent's decision process.

2.2. Training Problems

This process is subject to all of the common problems which face hill climbing algorithms. I used three techniques to battle these potential traps.

To prevent wild oscillations caused by overshooting an optimum configuration, the learning rate is set after each training cycle to either the default rate (0.4), or to the average error, depending on which is smaller. Thus, as the network converges, the error drops, and so does the learning rate. This has the effect of slowing the decent and getting a smoother convergence.

To avoid getting stuck in local peaks, momentum in the weights allows the system to surpass small obstacles. For more difficult obstacles such as plateaus and severe local peaks I used a technique I call selective simulated annealing. When a simple test determines that the learning has stalled, a corruption signal is sent through the network that randomly jostles the weights a little. This is usually an effective method to nudge the network out of stasis.

One final important thing to note is that preflop actions were filtered from the training data. By filtering

```
+---+-------+-------+-------+ +--------+
| * |   F   |   C   |   R   | | FREQ   |
+---+-------+-------+-------+ +--------+
| F | 0.13  | 0.0030| 0.0030| | 13.6%  |
+---+-------+-------+-------+ +--------+
| C | 0.0   | 0.584 | 0.033 | | 61.8%  |
+---+-------+-------+-------+ +--------+
| R | 0.0   | 0.105 | 0.141 | | 24.7%  |
+---+-------+-------+-------+ +--------+
+---+-------+-------+-------+ +--------+
| % | 13.0% | 69.3% | 17.7% | | 85.6   |
+---+-------+-------+-------+ +--------+
      Total Squared Error = 0.2477
```

Fig. 2. A Typical Confusion Matrix

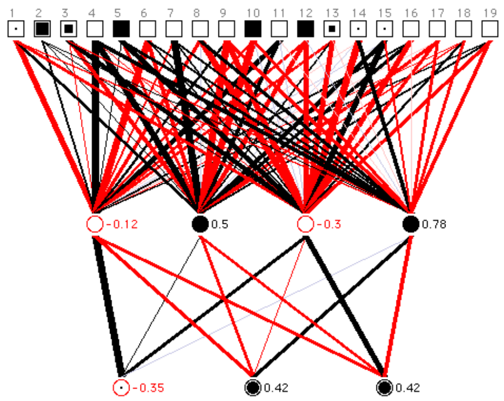

Fig. 3. A Neural Net before training.

preflop observations out of the training data and training only on postflop observations, the accuracy of the networks skyrocketed from an average range of 55-70% to 75-90%. This is probably due to the fact that preflop play has a much different style than postflop play, and thus hindered the learning process by forcing generalizations which could cover both stages of the game. Preflop actions are also much more difficult to predict since far less information is available than is in the postflop stage of the game.

2.3. Determining Accuracy and Average Error

Accuracy is determined by running each item in the test set through the network, and comparing the output with the correct result. If the node with the maximum value is the correct node, then this is treated as a successful identification. The error in the network is calculated as the average squared difference between the output and the answer.

A useful way to view the accuracy of a network is through a confusion matrix as shown in **Fig. 2.** Columns F,C,R represent the proportions in which the network predicted Fold, Call, or Raise. The F,C,R rows show the proportions of times the opponent actually Folded, Called, or Raised. The diagonal, then, is occurrences where the network predicted an action which was correct. All of the values in the 9x9 matrix sum to one. The FREQ column displays the percentages of actions for the player, and the bottom row shows the percentages for each prediction. The bottom left cell gives the percentage sum of the diagonal which is the overall success rate.

2.4. Constructing a Probability Triple

By normalizing the three output values of the neural network, a probability triple is ready for use. The resulting distribution represents the network's degree of belief that a player will take a certain action. If we have also calculated a confusion matrix, then it can also be used to bias the triple further. For instance, if the triple given by the network is {0.85,0.10,0.05} we will most likely choose 'fold' as the predicted action. However, we may be concerned from the confusion matrix that our network predicts folding when the player actually raises a significant portion of the time. This is discovered by examining the 'F' column. By adding the two triples (or performing some other operation which biases the distribution appropriately) we can get a new triple which represents a more conservative prediction. This new distribution is now biased by our knowledge that the network makes certain types of mistakes.

2.5. Graphical Representation of a Network.

As part of the Java based ANN class structure, the ability to graphically display the ANN was an easy addition. By visually displaying the entire network, debugging was greatly facilitated. Spotting a problem is much easier with a figure than dumping an array of weights to a terminal. Even more importantly, the network diagrams make it easy to see how the network changes over time and to clearly pick out the dominant weights and nodes. To some extent, it is even possible to see how the network works by looking at the relative sizes and direction (red is negative, black is positive) of the weights.

Fig. 3 shows a neural net diagram before training has occurred. All weights are randomly chosen values from -0.5 to 0.5. The square nodes at the top are the inputs, and the outputs are the circular nodes at the bottom. The number printed beside each node is that node's bias value (an overall weight for the node itself). The current input is shown by filling in an input node with black. A black square has a value of one, a white square is 0, and a partially filled square represents a real value between 0 and 1. The numbers printed above each input node correspond to the inputs listed in **Fig. 1.**

2.6. Baseline Comparisons

A	B	I	II	III	sb/h
218	361	63.43	69.53	90.03	-0.017
250	217	52.07	64.06	75.58	0.1314
1323	615	58.21	72.20	80.00	-0.076
237	116	56.03	72.41	75.86	-0.078
325	109	55.05	73.39	82.57	0.1267
90	322	51.24	70.19	82.61	0.1660
86	138	65.22	80.43	81.16	-0.138

Fig. 4. Seven different players trained and predicted.

```
   A) Number of training examples
   B) Number of testing examples
   I) Regulur opponent modeling accuracy
  II) Enhanced opponent modeling accuracy
 III) Neural Net Accuracy
```

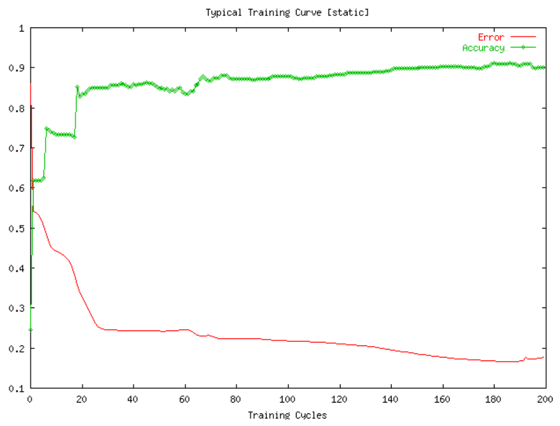

Fig. 5. A Typical Training Curve for a Static Opponent Modeling Problem.

To have a standard with which to compare the networks to, two simple predictors were created. The baseline predictor is identical to the opponent modeling used in *Poki* and *Loki*. Frequencies of an opponent's actions are stored in 12 different situations (combinations of the game stage and the number of bets the opponent had to call). The most frequent action in a given situation is chosen by the predictor as the action.

After studying the neural network's for identified significant features, I wrote a fancier version of the simple predictor which had a larger context set (stage, bets-to-call, last-bets-to-call, last-action).

3. Experimental Results

3.1. Static Opponent Modeling.

The standard test of the network's abilities is to take a stored file of training data and train the network on it. Next a separate file (of the same opponent in different games) is used to test the network's accuracy. It is important to use a separate set of training data and testing data. It is the only way to ensure that the network has learned to generalize from the training data. Without a test on separate data we cannot be sure that the network has not over-fit the data, essentially becoming a fancy look-up table.

Fig. 4. shows a chart of the results from seven different players of varying strengths. The last column 'sb/h' gives the player's IRC small-bets-per-hand statistic which is a simple measure of how strong they are. The simple predictor averaged 57% accuracy, the advanced predictor averaged 71% accuracy, and the trained neural networks accurately predicted 81% of a player's actions on average.

Fig. 5. Shows how the network's error and accuracy change after each training cycle (in one cycle the network is trained once on each training example). The accuracy typically improves in three steps (this is shown even more dramatically in **Fig. 9.**). It first learns to always choose the most frequent action overall. After it plateaus for a while, it eventually figures out how to make a distinction between two of the most common actions. Another plateau ensues after which a third distinction becomes possible and the network can recognize common situations for all three

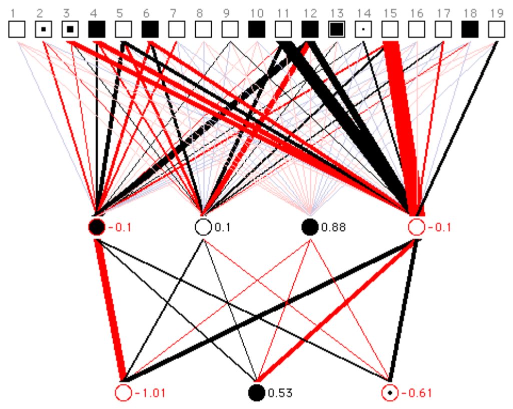

Fig.6. A Network after being trained on an opponent. (Shown correctly predicting a call)

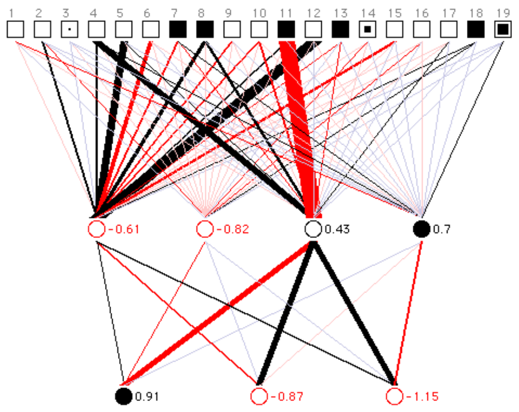

Fig.7. A Network trained by composite data, (Shown successfully predicting a fold)

```
+---+-------+-------+-------+ +-------+
| * |   F   |   C   |   R   | | FREQ  |
+---+-------+-------+-------+ +-------+
| F | 0.159 | 0.0070| 0.0   | | 16.7% |
+---+-------+-------+-------+ +-------+
| C | 0.0   | 0.63  | 0.014 | | 64.5% |
+---+-------+-------+-------+ +-------+
| R | 0.0   | 0.13  | 0.058 | | 18.8% |
+---+-------+-------+-------+ +-------+
+---+-------+-------+-------+ +-------+
| % | 15.9% | 76.8% | 7.2%  | | 84.78 |
+---+-------+-------+-------+ +-------+
      Total Squared Error = 0.2647
```

Fig. 8. A Confusion matrix of the composite network predicting a new player.

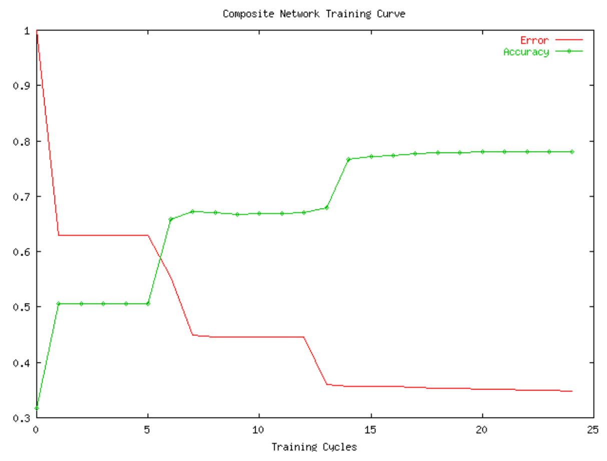

Fig. 9. Error and Accuracy during the training of a composite network.

actions.

3.2. Generalized Opponent Modeling

As an experiment, a 1397-item training set was created from six different player files to create a sort of 'default' player training set. To get a broad range, the set included weak players, moderate players and strong players. After just 25 training iterations, the network was able to correctly predict 117 of the 138 (85%) actions of a seventh player whose data was not included in the composite network. Further iterations yielded little further improvements. **Fig. 7** shows the final network. Note that a few input features have relatively strong influences, and that others have virtually no influence at all.

By viewing a trained network in this way we can easily spot what game features are highly correlated with an opponent's actions. From observations of many different player's networks, it appears that the current stage of the game, and to some extent the bets-to-call are only minor factors. This evidence suggests that the current method of opponent modeling with action frequencies (using just these factors) is flawed. If action frequencies were based instead on the more important factors (such as the last action), we could reweight opponent weight tables based on unanticipated actions. If we have an 85% expectation that a player will call, but they then raise we should be concerned that they have an unusually strong hand.

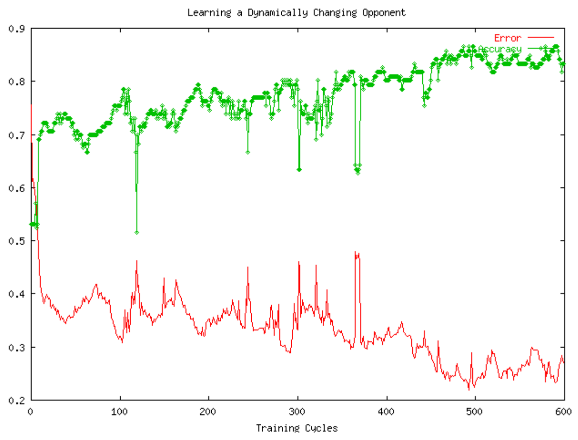

Fig. 10. Starting from a random neural network and training and testing with dynamically changing data.

3.3. Dynamic Opponent Modeling: Tracking a Moving Target

All modeling done for this project was done offline (using opponent data collected from IRC Poker games). To simulate an online learning situation an experiment was set up where new observations were slowly added to the data set and the accuracy was determined by the successes in predicting only the most recent actions.

The graph shown in **Fig 10.** reveals the network's struggle to keep on a moving target, but it manages. The curves are much more noisy, with sudden spikes where accuracy plummets for a brief moment. Overall, the network manages to do fairly well. After 400 cycles, the error takes a sudden drop of roughly 10%. It is possible that at this time the network has finally figured out some new factor successful in predicting the actions.

A big problem faced with any opponent modeling system is responding to sophisticated players who rapidly change their style of play to avoid being modeled. One way this can be dealt with is by training not one network per player, but several. All networks compete to be used, and are scored on their accuracy in predicting the opponent's most recent actions. The competing networks are trained in different ways. Some are trained with a full history, some with just recent events, others with a balanced mixture. A system such as this should give us the most accurate model possible at any given time.

4. Discussion

4.1. Future Work

The next step will be implementing a real-time online modeling system. The first important consideration is the amount of time it takes to get an accurate net. Since much of the time in a poker game is spent waiting while other players in the game make their actions, a low priority background thread can train networks for the players currently in the game. A default network can also be trained on all opponents to get a reasonable default model. The default model can be used until enough observations have been made to train a dedicated net for a player.

Once we have an accurate model of our opponents, what can we then do with this model? One possibility would be to run a series of simulations as is done in the current poker system. Instead of using our own betting strategy for simulating the opponents, we can now use our accurate networks. This should yield much more accurate results. For instance, if we have 85% accuracy on a player, we will be able to simulate with high certainty whether or not our bluff will be called or not by that player.

I am confident that the accuracy of the neural nets can be much improved upon. In the 19 input nodes used, there was very little board information, field aggressiveness data, positional information, and dozens of other potential featues. There are also many techniques for building better networks and for training them more effectively, which I have not yet tried. Future enhancements could improve these network's accuracy even further.

4.2. Action Frequencies V.S. Neural Networks

Obviously table based action-frequency predictors can do a reasonable job of predicting the opponent. The enhanced version which I wrote for this project achieved 71% accuracy on average and it could most probably be enhanced to approach 80% by continuing work on it. Tables have a huge speed advantage over neural networks — they require an insignificant amount of processing time. However, neural networks have a mass of developmental advantages over tables. The more context features we wish to add to a table, the more work must be done to properly combine all of the table entries when predicting things. First of all, the order in which things are stored matters. The tables must be organized from most significant and general features to the most specific. The more features, the more observations are required to fill the table with enough data to make any inferences. It can be a difficult task to combine the table entries in a way which properly uses both general course-grained information and specific fine-grained observations.

Neural nets on the other hand can be extended with more context features effortlessly,, the order does not matter, and the network can automatically generalize from a few observations as well as handle specific cases. Neural Networks handle all of the R&D involved in constructing tables.

4.3. Conclusion

Neural Networks have proven themselves to be a useful way to model poker players. These simple ANN's were able to predict opponent actions at significantly higher accuracy than the existing opponent modeling system. ANN's have the added benefit of being a useful tool for discovering the relative importance of different input features. By examining ANN's, a better action frequency based predictor was created.

5. Acknowledgements

I would like to thank Markian Hlynka for his help in debugging the back propagation algorithm and answering so many of my ANN questions. And thanks to Darse Billings for finding the pot-odds bug.

6. References

D. Billings, D. Papp, J. Schaeffer and D. Szafron, *Opponent Modeling in Poker*, 1998. AAAI, pp.493-499

M. Hlynka. *A Framework for an Automated Neural Network Designer using Evolutionary Algorithms.* 1997.

S. Russel, and P. Norvig. *Artificial Intelligence: A Modern Approach.* 1995. Prentice-Hall, New Jersey. pp. 563-596.