CS478 Final Project : Neural Network Blackjack Learning

James Landis

Monday, May 15th, 2000

Introduction

Every summer, Nick Chodorow's uncle takes his entire family to Las Vegas for a week. On the first day, he wins enough money on the Blackjack table to pay for the entire trip, including the money the rest of his family gambles away. There are many things that make him a successful gambler, and the most important is that he has a good strategy for playing the game.

Using a "good strategy", the chance of winning a given hand of Blackjack has been calculated to be about 47.5%. By counting cards, these odds can be increased to about 49.7%. Obviously, there must be better strategies that allow gamblers to consistently win money at Blackjack. Is it possible for a neural network to learn the ideal strategy for any given Blackjack hand? How good is the resulting strategy?

Following Gerry Tesauro's work with Backgammon, the field of neural networks looks very promising for learning other game strategies. The random nature of the dice rolls in Backgammon makes it very easy for a learning algorithm to explore a search space, since the learned strategy does not have complete control over what moves are made over the course of the game. Similarly, the randomness of card shuffling and dealing allows a representative data set to be constructed for Blackjack. Because of Tesauro's success with TD-Gammon and the random factor of both games, it is expected that a neural network will perform very well when learning a strategy for playing Blackjack.

Unfortunately, this may not be true. The performance of the studied network design was very poor. The network converged from a set of random weights on a set of output values whose average absolute error was generally larger than the expected outputs even on the training set! A different network design might help alleviate the grossness of the error, but it is more likely that more information is required to learn a good Blackjack strategy than the information used.

Problem Definition and Proposed Solution

Task Definition

The learning task was to train a neural network to output the expected winnings or losses of a given set of cards for three possible actions: hitting, standing, or doubling down. The input to the network should be the current game state, including the cards in the player's hand and the card the dealer is showing. For a card-counting strategy, the cards seen so far should be considered, as well, but this aspect of the problem was ignored. Different

casinos use different numbers of decks and shuffle at different times. Given the game state, the network should output the expected return of each possible action. Splitting was also ignored during this study because of the complexity of representing it in the training data. The odds of winning a split are also considerably less than those of the standing or hitting.

Training such a neural network is interesting because it can be used to develop "the perfect strategy". Assuming that the network can accurately predict the winnings of a given action for any given hand, the "perfect strategy" is optimal because it makes the move with the highest return every single time based on these predictions. If such a strategy can be developed and is simple enough to memorize, then Blackjack becomes a sure way to make money at the casinos.

Implemented Solution

The chosen neural network design used 26 inputs. The first 13 inputs each corresponded to one of the 13 ranks (2, 3, 4,…, K, A) and were used to represent the dealer's card. Each of the 13 inputs was 0 except for the input corresponding to the dealer's card, which was set to 1. The next 13 inputs also represented each of the 13 ranks, but were used to represent the player's cards. Again, the input was 0 if the player did not hold the card, but the weights for the other cards were adjusted according to the number of that card in the player's hand. For example, if the player held two aces, the input for the rank of ace was 2.

The network design called for three output values. Each of the three values represented the expected winnings of a specific action. The three actions (hit, stand, double down) each had their own separate output. The output values ranged from -2 to +2, corresponding to a loss of $2 or a gain of $2 when the initial bet was $1. The output nodes use the sigmoid squashing function to compress the data to the range [-1, 1], then multiply the output of the sigmoid by 2 in order to represent the full range of winnings and losses possible.

The network also had one layer of fully connected hidden nodes. Four different networks were trained, each with a different number of hidden units. The first used 20 hidden units, the second used 40, the third 60, and the last 80. Tesauro trained two networks, one with 40 hidden units, and one with 80 units. The performance improved with the number of units, but since his network had over 200 inputs, it was possible that the Blackjack learner could be adequately trained with only 20 hidden units since it is an order of magnitude smaller. Each of the hidden units also used the sigmoid squashing function to output a value in the range [-1, 1].

The network was trained using a backpropagation algorithm similar to the one defined in Mitchell. However, each training example gave the expected earnings of only one action at a time. To account for this, the training output vector was combined with the current outputs of the other two actions. The resulting weight update for each of the hidden node inputs to these other two outputs is 0, so effectively the algorithm only changes one of the outputs at a time. Instead of calculating these weight updates every time, the algorithm

simply skips the update of the weights between the hidden units and the other two outputs.

The Training Process

A Java class was built to create and train the neural networks using specified parameters: the number of hidden nodes, the number of training examples to use, and the file name for the training examples. The network weights were stored in files that were updated every 500 iterations through the training examples, in case of abnormal termination. First, the algorithm reads in the old weights from previous training and stores them in two arrays: the first contains all the input-hidden node weights, the second contains all of the hidden-output node weights. If a previous weight file does not exist, new random weights are generated in the range [-0.05, 0.05]. The file naming convention ensures that the correct weights are used. Each file is named "blackjack_ann_x_y", where x is the number of hidden units and y is the number of training examples. Had the training gone reasonably well, the performance of each network after the specified number of training examples on a test set would have been evaluated and compared.

Originally, the algorithm then iterated through the specified number of training examples, performing backpropagation on the network until a fixed minimum absolute error was obtained. This absolute error was an average of all of the absolute errors of each individual training example. However, the training never converged to an acceptable minimum absolute error. Because of this problem, the algorithm was modified to perform backpropagation until the change in the average absolute error fell below a given threshold.

Experimental Evaluation

Training Data

The training data was generated using a PHP3 script that output a newline-separated text file of training examples. Each training example was formatted as follows:

$$n\ d\ P_1\ P_2 \ldots P_m\ a\ w$$

where n is the numbered index of the training example, d is the dealer's card, $P_1 \ldots P_m$ are the player's cards, a is the action taken, and w is the winnings from that action. Each element is separated by a single space. The cards are represented by two characters. The first character is the rank of that card (2-9, T, J, Q, K, A) and the second is the suit (c, d, h, s). The training algorithm ignored the suit data, but future algorithms might take it into account. The action taken, a, was represented by a single character (h = hit, s = stand, d = double down). The winnings, w, ranged from -2 to 2.

The data was generated just as a real series of Blackjack hands would be played out. The dealer used three decks and shuffled each time the remaining pile was 10 cards or less.

The shuffling was indicated by a separate line with the text “-- shuffled --" in the data set. Each hand is dealt, then the player takes random actions until one of the following happens: the player stops hitting (stands or doubles down) or the player busts. The dealer then hits until his point total is 17 or higher. Finally, the winnings are calculated. Cases where the dealer makes a Blackjack are ignored, since the player automatically loses and no strategy is involved.

Each successive action in a hand is a separate training example. For example, consider the following data elements:

13 6c 6c Ks h -1
14 6c 6c Ks 2s h -1
15 6c 6c Ks 2s 3s h -1

This sequence represents a hand where the dealer is showing a 6 of clubs, and the player is dealt a six of clubs and a king of spades. The player hits three times, drawing first a 2 of spaces, then a 3 of spades, and finally busting with the third card, earning a loss of -1. Obviously an intelligent player would not have hit once he reached 21, but example 15 is necessary to teach the concept of not hitting once 21 is reached.

A total of 100,000 training examples was generated and used as a training set for each network. An initial training set of 1,000,000 examples was generated, but it was useless because of an incorrect hand evaluation function. Instead of valuing aces as 1 or 11, the algorithm used 1 always, and hence causes the game-playing algorithm to behave incorrectly. The original data set had to be thrown out.

Anticipated Analysis

Had the networks been trained within a reasonable threshold of error, they would have been evaluated based on their performance on a given test set. The effects of the training set size and the number of hidden nodes on the performance of the network were of particular interest. Since each network was trained using the same training set, but variable numbers of training elements, the comparisons would have been particularly meaningful. However, the networks were essentially useless at predicting the expected outcomes of a given play, since the average absolute error was always greater than 1 for every single network.

Analysis of Trained Networks

As shown in Table1, the average absolute error over the training examples for each of the networks was greater than 1. This means that if the network predicted earnings of 0 for a given action, the action would on average actually yield +1 or -1. It would not be wise to base a strategy based on an evaluation function with such a low precision.

		Training Examples				
		1000	5000	10,000	50,000	100,000
HN	20	1.18615	1.21703	1.21152	1.20758	1.20619
	40	1.59600	1.63740	1.63430	1.63150	1.62589
	60	1.80201	1.86820	1.85390	1.84102	1.83273
	80	1.80200	1.80700	1.80570	1.81134	1.81293

Table 1 - Summary of Absolute Error of Various Network Training Parameters

Unlike Tesauro's TD-Gammon, the Blackjack networks did not improve in their accuracy on the training data with increased hidden nodes. In fact, the approximations were worse when more training hidden elements were used, with the exception of a slight improvement with 80 hidden nodes. The network error did not change significantly with increased training examples. In some cases, it even increased with increased training.

Reasons for Failure

There are many possible reasons that the neural networks failed to learn the concept of playing Blackjack. The most likely reason is that the game state cannot accurately be represented with just 26 inputs. The distribution of the outputs according to the inputs makes it difficult for the network to learn the game concepts. Another less likely possibility is that the training was caught in local minima as a result of a skewed learning rate or poor initial weight choices.

A representation of the game state that only includes one of the dealer's cards has some obvious drawbacks. If the dealer is showing a face card, his hand can still be one of two extremes: he can be holding 20, which is very difficult to beat, or holding 16 and is very likely to bust. No matter what cards the player holds in his hand, the deciding factor is very often the second card in the dealer's hand. A representation that does not take the second card into account thus makes it very difficult for a network to learn the game concept. There may be thousands of examples with the dealer holding a face card, yet the outcomes can be distributed relatively evenly among the values [-1,0,1] (ignoring doubling down). The best the algorithm can do is to converge to output 0, where a two thirds of the time it will be off by 1. This explanation fits very well with the performance of the trained networks when the losses due to doubling down are taken into account.

One of the drawbacks of ANN backpropagation learning is that it is susceptible to finding local minima in the target function during gradient descent. Because of this susceptibility, another possible reason for the networks settling into erroneous error minima is that the original network weights caused all of the networks to find one of these minima immediately. This also seems to fit the results, because the networks settled into very

similar minimum errors. A less likely problem is that the learning rate was not sufficiently large or small to allow the algorithm to find the global minimum. However, several different learning rates were tested, ranging from 0.00005 up to 10, all with similar results, so this was at least not the sole reason for failure.

Conclusions and Recommendations

Although this study showed limited success of neural networks' learning of Blackjack, this does not mean that Blackjack cannot be accurately learned by a neural network. There are many ways in which the studied networks can be improved. Most importantly, an appropriate representation of the Blackjack game state should be developed. A statistical analysis of the training data might suggest a way to partition the input so that it is more linearly separable. Consideration of previous cards might also lead to increased accuracy. Even altering the structure of network so that it outputs only one value and adding boolean inputs to represent each of the possible moves might improve the performance.

In order to eliminate some of the pitfalls inherent in neural network learning, the backpropagation algorithm might be updated to use momentum. This would limit the tendency of the algorithm to settle into local minima or to get stuck on flat plains in the target function where very little change in the error occurs. A study of varying initial weights might also yield some information about the ability of the network to learn target functions.

References

Scoblete, Frank. Beat Blackjack. New York: Bonus Books, 1996.

Sutton, Richard and Barto, Andrew. Reinforcement Learning: An Introduction. Cambridge, MA: MIT Press, 1998.

Tesauro, Gerald. “Practical issues in temporal difference learning.” 1992.