A Machine Learning Approach to Playing Rack-O

Chris McNeill, John Nuttal, Isaac Nygaard
CS 478, Winter 2014
Department of Computer Science
Brigham Young University

Abstract

A machine learning approach was taken to create an AI to play Rack-O. We initially started with a learner that would play randomly against itself and save the moves it used when it won a game in a good amount of time. We then moved to creating more intelligent AIs using different methods such as temporal difference and Q-Learning.Our new AIs performed significantly better and were able to beat human players.

1 Introduction

Rack-O is a game that involves two or more players wherein they draw cards and place them in a rack. The object of the game is to place the cards in ascending order. A player initially draws cards from a shuffled deck and places them in a rack starting from the bottom and proceeding upwards. This results in a rack that has the cards out of order. The cards each have a unique number on them and are labeled 1-N where N is the maximum card value, usually 40 to 60. Each rack usually has 10 slots but more or fewer slots can be used. Players take turns drawing from either the deck or the top of the discard pile. The top card of the discard pile is visible to all players. Once a player has drawn a card, he or she uses it to replace a card in his or her rack. The card that is replaced is placed on the top of the discard pile. If the player does not wish to use the card he or she drew, he or she can discard it. The game ends when one player has a rack where all the cards are in ascending order from bottom to top.

1.1 Motivation

We thought that a machine learner could easily record its moves, have them scored by some score metric, and learn to use the best moves in future games. With enough scored moves as training data, the machine learner could become quite intelligent. We decided to start with a game where the racks had five slots and the cards were numbered 1 through 30. This would reduce the problem enough that our data instances would not be too big, but the problem was still big enough to be learnable by a machine learner.

2 Methods and Data

2.1 Data Sources

Our original plan was to create an AI that played randomly against itself for a multitude of games. When the learner would win, it would save its moves as good moves and score them based on how many moves it took to win. Fewer moves, which would indicate more intelligent move choices, would result in a higher score. If the losing player was close to having a winning rack, his rack would also be scored, although not as well. This would become our initial training data.

Once enough random games had been played, the highest-scoring training data would be run through a backpropagation learner. The now learning AI would continue playing games against the AI and would continue to score its new moves. This would result in a type of bootstrapping learning model.

Once we created this Random-Then-Bootstrapping AI, we played it against a simple random player. Our AI player would consistently beat the random player. However, when played against a human player, our AI almost never won.

2.2 Data Instances

We needed two data sets. First, we needed a data set for deciding whether to draw from the deck or from the discard pile. Second, we needed a data set to help decided where to place the card in the rack (or whether to discard the drawn card because it would not be beneficial in any slot of the rack).

The drawing data instances initially contained the player's current rack, the probabilities of drawing a card higher than the card in each slot in the rack, the probabilities of drawing a card lower than the card in each slot of the stack, and the current card at the top of the discard pile. The output value was whether to draw from the draw pile or from the discard pile. The play data instances contained the current rack and the card drawn. The output value indicated the slot the drawn card should be placed in or to discard the drawn card.

We used these features in our data sets because we hoped they would be enough information in order to make a good

decision. The probabilities were calculated using cards in the current player's rack and cards that had been seen by the player. For example, if the card in slot 1 was a 10, the base probability of a card lower would be 9/(number of cards in the deck). If the player knew that cards 2 and 5 had already been used, then the probability would drop to (9-2)/(number of cards in the deck).

slot1_pLower	Probability of drawing a card lower than the current card in slot 1.	.67
slot1_pHigher	Probability of drawing a card higher than the current card in slot 1.	.00
slot2_pLower	Same as above for slot 2.	.02
slot2_pHigher	Same as above for slot 2.	.58
slot3_pLower	Same as above for slot 3.	.11
slot3_pHigher	Same as above for slot 3.	.38
slot4_pLower	Same as above for slot 4.	.52
slot4_pHigher	Same as above for slot 4.	.31
slot5_pLower	Same as above for slot 5.	.56
slot5_pHigher	Same as above for slot 5.	.29
slot1	Card in slot 1.	29
slot2	Card in slot 2.	4
slot3	Card in slot 3.	8
slot4	Card in slot 4.	19
slot5	Card in slot 5.	17
discard	Card at the top of the discard pile.	23
drawDiscard	Output class T=draw from discard. F= draw from deck.	T

Table 1 – A sample draw data instance.

slot1	Card in slot 1.	2
slot2	Card in slot 2.	22
slot3	Card in slot 3.	15
slot4	Card in slot 4.	23
slot5	Card in slot 5.	29
drawn	The card that was drawn.	5
slot	The slot where the drawn card will be placed. 1 = slot 1, 2 = slot 2, etc. 0 = discard.	2

Table 2 – A sample play data instance.

2.3 Models

We chose a backpropagation model to learn this task. We chose backpropagation because all of the features were real-numbered values. We also chose backpropagation because the training could be done separately from playing, the weights could be saved once it had trained sufficiently, and the weights could be loaded into a model for game play. It would also be able to calculate moves quickly due to all the training being done *a priori*.

Our backpropagation model was capable of deep learning. The neural net would play against another AI in a series of games. When a predetermined number of epochs was reached without improvement (normally 100), the neural net would add another hidden layer and continue training. This procedure continued until a sufficient win percentage was reached.

3 Initial Results

We used several measurements when evaluating the performance of our model. The first measurement was the number of moves the AI needed in order to finish a game. Ideally, the maximum number of moves would be the number of slots in the rack. If the rack started out with each card needing to be replaced, then each card should only need one move to replace it. However, most racks start with at least one card in a useable position so the average number of moves over all games should be slightly less than the rack size.

Another performance measurement was the percentage of games won against a random player. Since the random player just randomly draws from the deck or discard pile and chooses a random slot for the drawn card, the player is essentially performing a Bozo sort which should finish in n! moves (where n is the number of slots in a rack). Any player with a slightly smart heuristic should be able to beat the random player.

The last performance measurement we used was the percentage of games won against another AI. This AI could be another instance of the AI or a completely different AI that does not rely on machine learning. This would be the true test of our machine learning AI. We also used percentage of games won against a human as a performance measurement but not as much as the previous three.

We had our AI play 1 million games against another instance of itself. At first, the two AIs were evenly matched in win percentages. They initially took around 1200 moves to finish a game. After 28 epochs, the first AI began to overtake the second and the average number of moves to complete a game dropped to 400. By the end (3333 epochs), one instance of our AI had won 64% of the games and was finishing in 113 moves. While the improvement was impressive, the number of moves per game was still not what we were hoping for.

We added the deep learning capability to the AI. Overall win percentage increased to 93% while the number of moves per game stayed at about 117 after 1188 epochs. When allowed to run longer (4230 epochs), the win percentage decreased to 85% but the number of moves per game dropped to 36.

When played in a tournament between a temporal difference AI, an AI that maximized the length of the longest usable sequence in its rack, our AI, and a random player, our AI won 54% of the games and had an average of 4.09 moves. However, a human could still beat our AI every time.

4 Model and Feature Improvement

Unsatisfied that our AI could not compete against a human, we began examining our model and seeing how we could improve it. We implemented a fairly smart AI that we found on the Internet dubbed "Kyle." We decided to use this AI as our benchmark. We then proceeded to try different learning approaches.

4.1 Baltar and Q-Learning

We created a Reinforcement Learning model using Q-Learning called "Baltar." This model kept track of a set of states, as well as a set of actions that can be performed for each state. Each action had an expected reward associated with it. All these rewards were initially zero.

Each turn, the reward function for the action taken on the previous turn was updated. This was done through a Q-Learning function, taking into account the highest reward for any possible action from the current state and the number of times that the particular reward has been updated. In order to explore the state space as much as possible, while training, it chose whatever action it has chosen the fewest number of times before.

Initially, when coding up Baltar for the first time, we chose a simple metric for dividing the states so that it would be able to explore the entire state space fairly quickly. For each card in the rack, as well as the card from the discard pile, we figured out whether it would be more likely to draw a card above or below that one. This gave a small state space of just 64 states for drawing and 64 states for playing. We expected this to be able to do better than random, but not by much. When we ran it, though, we were surprised by the variability in results it displayed. The final trained models were consistent in the percentage of games they won against random opponents, but there was a lot of variance between the models. They varied anywhere from winning only 2% of games against a random opponent to winning 80% of the time. To deal with this, after training a model, Baltar played 2000 games against a random opponent. If the model did not win at least 70% of the time, it discarded the model and trained a new one. This allowed the end result to be much more consistent.

We also tried training against Kyle, but found that models trained in this way only won about 20% of the time against a random opponent. This was probably due to the fact that the random opponent won so quickly that Baltar did not have sufficient time to explore and encountered only negative outcomes.

We then experimented with different metrics for determining states. When we tried creating one state for every possible configuration of the rack and drawn card, the program ran out of heap space. Other metrics that were tried, such as determining which sixth of the entire range each card was in or determining for each card whether it was 25%, 50%, 75% or 100% likely to draw above or below it. To our surprise, these did not perform as well as our initial metric. They consistently won less than 50% of the time against a random opponent.

We then started saving the best models to files and created an AI that read them in and played using a weighted voting approach, where each model would get a vote proportional to the percentage of games it won against the random AI. This AI won 75% of the time against the previous version, which had only one model.

4.2 Diablo

Another model we tried was one that we named "Diablo." Diablo uses a learned scoring function to decide what moves it should make. The scoring function gives higher values for racks that are more likely to win.

Decisions for each turn follow this algorithm:

• Drawing: Diablo looks at the card on the top of the discard pile is. It inserts this card into each slot of its rack, scoring the rack at each position. The slot that gives the maximum score is returned. If this improves the rack's score by a large enough margin, the top of the discard pile is drawn. Otherwise, a card is taken from the draw pile.

• Playing: The card that was drawn is inserted into each slot of the rack. The slot that gives the maximum score is returned. If this improves the rack's score at all, the move is played. Otherwise, the card is discarded.

4.2.1 Temporal Difference

Initially, we set the required margin for drawing from the discard pile at 1/(rack_size*2). To learn a scoring function, temporal difference reinforcement learning was used. Diablo uses a similar approach to the strategy Gerald Tesauro used in his famous TD-Gammon program. Basically, the scoring function tries to predict what the rack's score will be in one move. The score of the current move is set as the target output for the inputs of the previous move.

Error(t) = score(inputs(t+1)) – score(inputs(t))

At the end of a round, the target output is set to the score the player actually received.

Error(tend)=final_score_for_game - score(inputs(tend-1))

A neural network was used as the model for learning. Training was done at the end of each round. More influence was given to error values closer to the end of the game. Specifically, the weighting function for the error values followed this formula:

Weight(t) = max(0, (t-tend)/(40*rack_size) + 0.1)

Initially, the cards in each slot of the rack were used as input features for the neural network. Nevertheless, after training for one million games, it could only win 51% of games against a random player. These input features were augmented with probabilities for drawing higher and lower than a card in a slot but, the added features still did not improve performance.

Due to the ill performance of the neural network, we decided to replace the features with a number of high level, pre-computed features. The new features are calculated by computing scores for various criterion, namely:

• Distribution of cards in the rack

• Usable sequence length

• Probability of filling holes in a sequence

• Density of clumps in a sequence

We define a "usable sequence" as a set of cards that could be used together in a winning rack. Clumps are cards in a sequence that are adjacent to one another, and holes are slots that are not part of a usable sequence. A usable sequence is highlighted in green in Figure 1.

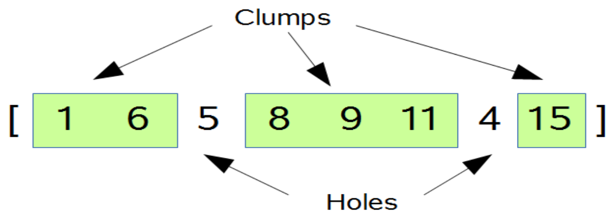

Fig. 1 – Clumps and useable sequences.

For a rack that is completely sorted, there are 2^rack_size number of usable sequences. The majority of the new features are computed from a usable sequence. Since the score for each of these sequences can be different, we need to run a separate set of features through the neural network for each possible sequence. The maximum score out of all usable sequences is returned as the score for the rack.

4.2.2 Diablo's New Data Instances

Table 3 shows Diablo's new feature set and explanations for each feature.

Turns	A typical rack can be sorted in about as many moves as there are slots in a rack. If this feature is closer to one, there is a high probability that the round will end. If the player's rack isn't close to winning, they may want to play for a high point score, instead of a win.
Points	Could be used to optimize for a high scoring rack.
Sequence Length	A sequence length equal to the number of slots in a rack will win the game.
Rack distribution error	Measures the error of each card, given a "flat" target distribution. The flat distribution is given by the formula: y = x(max_card-1)/rack_size + 1, where y is a card number (from 1 to rack_size) and x is a slot number (from 0 to rack_size-1). Without knowing information about probabilities or the rack's usable sequences, a flat distribution maximizes the chances of getting a sorted rack.
Clump distribution error	Measures the error of cards in clump centers, given a "flat" target distribution (described above). This gives a similar result to the rack distribution error, but doesn't penalize for cards that couldn't be used in a winning rack. The error of each clump is weighted by how large the clump is.
Probability	Gives the probability of filling holes in a sequence, if the player drew from the draw pile. Higher probability is better.
Average probability	In some cases, the probability of filling a hole is zero, making the entire probability score zero. However, there is a chance an opponent will discard a card for that hole or the player will draw such a card when the discard pile is reshuffled. The Average probability score averages probabilities for each slot, to account for this.
Adjacent density	Computes the differences between adjacent cards in a clump. If the density is one, the clump is in perfect sequence (1,2,3), whereas a density of zero has a very large spread (1,15,25). Higher densities in clumps will give higher probabilities of filling in holes.
Adjacent density penalized	This is the same as adjacent density, only it penalizes clumps with only one card in them. This may encourage the player to get larger clumps, rather than many clumps with only one card.
Center density	Calculates the differences between cards in a clump and the center of the clump. Adjacent density really only improves when the edges of a clump come closer together. Center density will improve if the interior of a clump comes closer together.
Skewed features	All the features except turns, points, and sequence length are included again, with their values weighted to a skewed distribution. The skewed distribution is linear, given by the points (0, 1) and (rack_size-1, 0). Racko is scored by counting the number of ascending cards, starting with the bottom and stopping when a descending card is encountered. If a player doesn't win the round, it may be beneficial to get the highest non-winning score, so the loss is not so detrimental.

Table 3 – Diablo's new feature set.

Since computing scores for all 2^rack_size number of usable sequences would take too long, we chose to compute scores only for sequences that were not subsets of another sequence (e.g. given sequences [1,2] and [1,2,8], [1,2] would be discarded). Later experiments showed that using all 2^rack_size sequences did not give any noticeable improvement.

The neural network was trained for 476,000 games, using a rack size of 10. The number of turns per round was limited to 1500. After 15,000 games, Diablo was able to finish a game under the 1500 move limit. At 43,000 games, Diablo's learning accelerated until it reached 10 moves per game. By the end of training, Diablo averaged 9.87 moves per game. Figure 2 is a graph that shows how Diablo's move count improved over time.

Adding a second hidden layer to Diablo's neural network with 30 nodes improved its performance further, giving 8.88 moves per game. A tournament between Diablo and Kyle resulted in 66.83% wins with only one hidden layer, and 71.4% with two hidden layers. We did not experiment with more than two hidden layers.

Experimenting with the threshold margin for drawing from the discard pile gave slight improvements. A good value for the margin was found to be 1/(rack_size*2.8169). This improved the win percentage against Kyle to 73.21%.

We tried using a neural network to automatically learn the best threshold, but this approach only degraded performance. We also experimented with a simple voting ensemble using Kyle, Max, Baltar, and Diablo. However, the ensemble was only able to win around 15% of its games against Diablo.

4.3 Casandra

We tried creating a model that would record the moves made by another model, such as Baltar or Diablo, and using

these moves as initial training data. The model would then play against another AI and would remember the moves it made when it would win. We dubbed this mimicking model "Casandra." Unfortunately, Casandra did not fare well, even against a random player. Most of the games would end in ties by reaching a predetermined number of moves without any player winning. Of the games that did not end in a tie, Casandra would win about one-fourth of the games. Deep learning caused Casandra to improve faster but it would still not beat the random player.

We concluded that the reason for Casandra's failure was that the initial training features that we were using, which Casandra would record from the other AI and use in later games, were not useful in deciding where to draw from and where to play the drawn card. Perhaps a neural net would be able to learn to play Rack-O if we could find better features.

5 Final Results

We created a 1 million game tournament consisting of racks of 5 slots. The three players were Kyle Diablo, and Baltar. Kyle won 37.8% of the games in an average of 2.86 moves. Diablo won 54.27% of games in an average of 2.76 moves, and Baltar won 7.9% of games with an average of 2.61 moves. Both Diablo and Baltar improved the number of moves per game over our initial AI's 4.09 moves per game.

The real test of our AI's intelligence, however, was the number of games it could win against a human player. Final version of Diablo won 45/100 rounds against a human player. This was a vast improvement over our initial AI which could not win any games against a human player.

It seems that Diablo's temporal difference learning model along with a new and improved feature set led to its dominance over the other AIs that we tested.

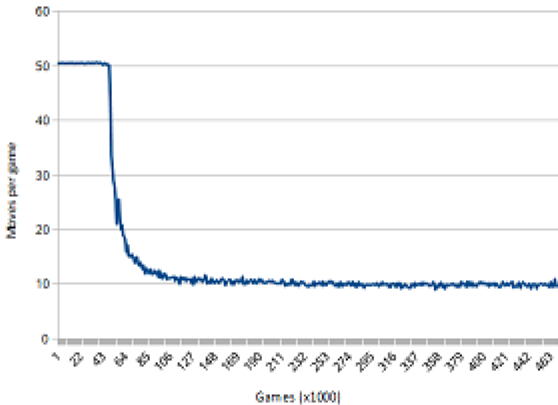

Fig. 2 – Diablo's moves/game over time.

6 Conclusions

Replacing our initial random-then-learning AI with Diablo led to a vast improvement in the number of moves and percentage of games won against both a random player and other AIs such as Kyle. Adding more hidden layers to Diablo also significantly improved its performance over Kyle. Diablo was also able to finally defeat a human player.

7 Future Work

If time were permitted, we would like to have attempted to develop Casandra so that she could successfully mimic another AI. We could try to have her learn using Diablo's data instances. For Baltar, we could continue experimenting with different state spaces to see if we could find one that would work better than the one it currently uses. We could also allow Diablo to train even longer and hopefully be able to defeat a human opponent almost every time. Due to the random nature of the game, victory is not always possible. Perhaps an ensemble of many instance of Diablo, all with different weights, would be more successful than the ensemble we mentioned in section 4.2

References

[Abbeel, 2013] Pieter Abbeel. Lecture 10: reinforcement learning. https://www.youtube.com/watch?v=ifma8G7LegE. 2013. Last accessed April 2014.

[Abbeel, 2013] Pieter Abbeel. Lecture 11: reinforcement learning II. http://www.youtube.com/watch?v=Si1_YTw960c. 2013. Last accessed April 2014.

[Tesauro, Gerald, 1995] Gerald Teauro. Temporal difference learning and TD-Gammon. Communications of the ACM, 38(3):58-67, 1995.

[Thompson, Kyle, 2008] Kyle Thompson. Rack-O revision 105. http://subversion.assembla.com/svn/rack-o/. 2008. Last accessed April 2014.