Using Artificial Intelligence to solve **Gin Rummy** Bachelor project

Name: Tsoe Loong Li
student number: 0223948
Study: Computer Science mono-discipline Informatics
Major Informatics/minor business science
Supervisor: Dr. W. A. Kosters

26th October 2006

Preface

For the graduation of the bachelor program Computer Science, I try to use Artificial Intelligence to solve the game Gin Rummy. Under supervision of Dr. W. A. Kosters, this paper is written in 2006, during the third year of my bachelor program.

The main idea behind the use of Artificial Intelligence is to make a program that has a performance close to good players or even performs better and faster. And in order to avoid hardcode programming, just try to let the program "learn" itself.

Abstract

Gin Rummy is a 52-card game, with not many rules and restrictions. The players have lots of freedom to play the game the way they wants to. Using brute force to calculate all possibilities is not an option since it will cost too much time and is practically not possible. A more clever way has to be used to handle this problem. The research question is: Can we create a competitive algorithm using a neural network to solve Gin Rummy? In this paper, we will see what the complexity is of Gin Rummy and how we are going to deal with it using a neural network. We will discuss what the results are, and how well the program is doing compared with skilled human players.

Key words: neural network, reinforcement, evolutionary learning, co-evolution, Gaussian noise, cross-over.

Contents

1 Introduction and game rules

In this paper, we will see if an artificial neural network is suitable for solving Gin Rummy, and if it is suitable, how to use the neural network to solve it. In Section 1, a short introduction about the history of Gin Rummy will be given together with a glossary explaining several terms used in the game and in this paper. Section 2 is about the complexity of Gin Rummy. Section 3 explains how a neural network works, how the research is conducted, and mentionsproblems and solutions during the implementation. Section 4 gives the trainings results and finally we have a discussion in Section 5.

1.1 Gin Rummy game history

As a game, Gin first appeared in the beginning of the 20^{th} century. Gin Rummy is a two-player card game, played with 52 cards. Some credit for the game is given to Elwood Baker (also a Bridge tutor) who invented Gin Rummy in New York. He later achieved much posthumous fame as a victim of an unexplained murder case. Only in the thirties of the 20^{th} century the game of Gin became popular, especially in the American culture as gin became a game of popular Broadway and Hollywood stars and featured in many movies as well [10].

The popularity of Gin is due to the fact that the game is quite easy to learn and it is a rather fast-paced game as well. The earliest form of the Gin Rummy can be traced back to the mid-19-th century Mexican game of Conquian. Conquian was a much simpler form of Gin Rummy with win-or-lose game played for a fixed stake. The best contribution that Elwood Baker invented to make gin what it is now, is to refine the scoring system, making it much more interesting to play for money or just for fun [10].

1.2 Task environment of Gin Rummy

The reason why I choose this as a bachelor project is partly because the game rules are easy to learn, and because the game allows for multi player interaction, which is more challenging than other games where only one player is actually playing. And it is an interesting task environment. Gin Rummy is partially observable, you don't know which card your opponent has in his/her hand. It is also stochastic, the next state of the environment is not completely determined by the current state and the action executed by the agent. One cannot know what card he/she will get from the draw pile. What makes Gin Rummy even harder is that it is also sequential. The current decision could affect all future decisions, short term actions can have long-term consequences. On the other hand the static environment of Gin Rummy makes it a bit easier for agents to do their work. The situation doesn't change when the agents is "thinking" and it is discrete. Last but not least, Gin Rummy is played in a multi agent environment. You have to consider in each step what the other agents (players) will do or react [8].

At each step/action, the program gets some feedback how good or how bad the move was, but this is only the feedback on short-term. What seems to be a good move now, might actually be a wrong one later. This makes it harder to use the reinforcement learning, which will be discussed later in the paper. The reasons mentioned above are the reasons, why I think Gin Rummy is so interesting to deal with.

1.3 Glossary & Game Rules:

Gin rummy is a two-handed card game that can be summarized as follows [9]:

- Deck: standard 52 card deck.
- Rank: King=highest, Queen, Jack, 10, ..., 2, Ace=lowest.
- Points: King, Queen, Jack = 10, Ace = 1; all others = face value, which means 10=10, 9=9, etc.
- Deal: distribute cards to the players, 10 cards to each player.
- Discard pile: pile of cards that are removed from a player's hand. The discard pile is always face-up.
- Draw pile: remaining cards after the cards are being dealt at the beginning. The draw pile is always face-down.
- Meld: a set of cards with he same value or sequences of 3 or more consecutive cards of the same suit.
- Deadwood: the total face value of the remaining cards that can not form melds.
- Goal: form melds and reduce deadwood as much as possible; a single card can not form part of a set and a sequence in the same hand. A goal state is a state with at most 10 deadwood points.

- Turn: during each turn a player can take the top card from the discard or draw pile, must discard one card face-up on the top of the discard pile and, if a goal state is reached, may lay down meld and deadwood (called *knocking*).
- Knock: to end the game by exposing one's hand. Melds will be formed and deadwood will be counted to see which player wins. A player can not knock when he/she has more than 10 deadwood points.
- Gin: to end the game with no deadwood points.
- Play: players alternate turns starting with the dealer's opponent until one player knocks or gin.
- Laying off: after one player knocks, the opponent may extend any of the knocking player's sets or sequences (called *laying off*) with any of his/her deadwood.
- Undercut: when after laying off, the opposing player's deadwood points are equal or less than the player knocking.
- Score: player who knocks scores the difference between the other player's deadwood points and his/her own. If the player who knocks has no deadwood, the other player is not allowed to lay off, and the player knocking receives a score of 25 plus the other player's deadwood points. In case of undercut, the opponent scores 25 plus the difference in points instead of the player knocking.

Due to time constraints, a couple of simplifications to the game have been made for this experiment. For example the laying off option is not implemented and the feature that the game stops when one of the players has a score 100 is not implemented as well. The maximum deadwood count is fixed at 10 before a player can knock. The 25-point bonus as well as the penalty is for simplicity not implemented.

At the beginning, each player gets 10 cards which are unknown for the opposing party. Let us say player 1 starts first. Player 1 has only the option of drawing from the draw pile since there is no discard pile yet. After drawing a card, player 1 has to throw one of the eleven cards away to the discard pile and pass the turn to player 2. Now the discard pile is no longer empty. Player 2 has the option to draw from the drawing pile as well as the discard pile. After this, player 2 has to throw a card of his/her choice to the discard pile. From now on, each player in turn takes either from the drawing pile or the discard pile and then adds a card to the discard pile. A player can only takes the top card of the discard pile. Any card below the top card is forbidden to draw.

The goal is to group the cards in one's hands into melds. We denote

$$\begin{aligned}\spadesuit &= \text{spades}\\ \heartsuit &= \text{hearts}\\ \clubsuit &= \text{clubs}\\ \diamondsuit &= \text{diamonds}\end{aligned}$$

Suppose we have ♠8-♡8-♣8, ♣3-4-5-6, D9, ♠ace, ♣queen in our hands. In this case, we have 2 melds. One set of cards of the same rank (8) and one set of consecutive cards of clubs. The cards 9, Ace and Queen are unmelded cards in this hand. This hand thus has a total of $9 + 1 + 10 = 20$ deadwood points.

Players continue to draw and throw cards until a player knocks or announces gin. A player can only knock when he/she has less than 10 deadwood points. Knocking is not obligatory; the player can decide to play further even if his or her hand has less than 10 deadwood points. After knocking or announcing gin, melds are formed with as few deadwood points as possible. The player with the most deadwood points loses the game. If a player knocks with no deadwood points, it is called a Gin. If none of the player has knocked and there are no more cards in the drawing pile, then the game is stopped and deadwood points are being counted. This is seldom the case, since of the games stopped before 20 turns generally. In practice, this rarely happens, since most of the games stop at 20 turns.

2 Complexity of Gin Rummy

Although the game Gin Rummy is played only with a simple pack of playing card which has only 52 different cards, it is quite hard to calculate all the possibilities.

Gin Rummy is partially observable, stochastic, and sequential; its multi agent environment makes it even more challenging. One cannot know what the opponent player has in his/her hand. You can only see what hand you have got and what card is at the discard pile. Along the game, you can guess what your opponent has in his/her hand, but the game only uses 52 cards, so it will be reshuffled very soon.

Second, your opponent can change his/her strategy along the game, so you have to adapt your strategy as well. Let's take a look at the decision tree of Gin Rummy (see Figure 1). Assume player 1 is to play. Then player 1 can choose to pick a card from the discard pile or pick a card from the draw pile. Let's assume player 1 picks a card from the discard pile, then he has 11 choices to play. Now follows player 2; player 2 can also pick a card from the discard pile or from the draw pile. Let's say player 2 picks a card from the discard pile; although player 2 has only 11 cards now, there are 41 possible cards played from the viewpoint of player 1. Player 1 only has knowledge about the ten cards, which he is holding in his hands, and the one card that is just played. From the point of player 1, player 2 can play one of the 41 possible cards which are unknown for player 1. After player 2 has played a card, player 1 follows. Once again player 1 have 11 cards. However, the unknown cards are reduced with two. Each time when player 1 has to throw a card, the unknown cards for player 1 are reduced with two: one card that is thrown away by player 1 in the previous turn and one card which player 2 just played. Then player 1 can play again and again with 11 possible moves, and so on, see the decision tree below. (The decision to draw form the discard pile or draw pile is for simplicity not shown in the tree.)

As shown above, the tree can expand very fast and very wide. In a normal Gin Rummy game, it usually reaches 20 hands before the game ends. Assume the Gin Rummy game ends at hand 20, then the number of possible states is:

$11 \cdot 41 \cdot 11 \cdot 39 \cdot 11 \cdot 37 \cdot 11 \cdot 35 \cdot 11 \cdot 33 \cdot 11 \cdot 31 \cdot 11 \cdot 29 \cdot 11 \cdot 27 \cdot 11 \cdot 25 \cdot 11 \cdot 23 \approx 2.47 \cdot 10^{25}$

A card can be:

- In player's hand (IPH), so player 1 has it

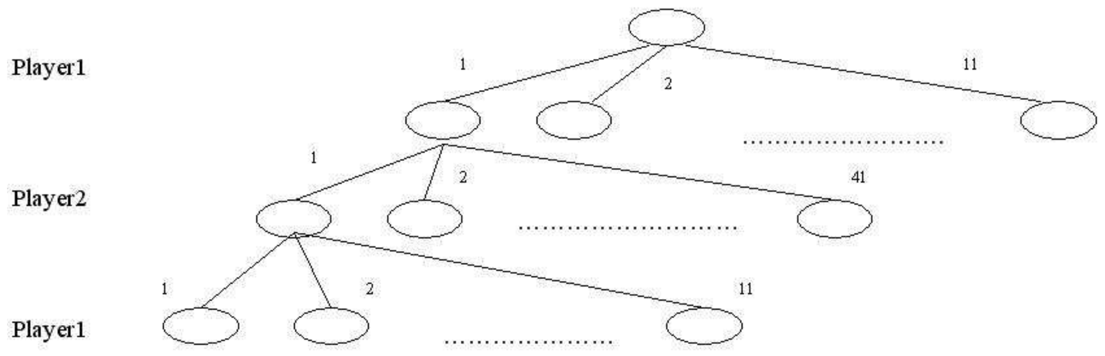

Figure 1: Decision tree for player 1

- In opponent hand (IOH), the card is in player 2's hand
- In top of discard pile (TDP), the card that has just been thrown away
- In discard pile (IDP), the card is being thrown away several turns ago or
- Unknown (UNK), no knowledge where the card is. This means the card is obviously not in the hand of player 1, but player 1 does not know where it is. It can be either in player 2's's hand or still in the draw pile.

A card is considered to be in the hand of the opponent if the agent notices that the opponent actually takes the card from the discard pile and has not thrown it onto the discard pile. Any other card is considered unknown (UNK). So with 52 cards, each card can be in five states, and we have at most 5^{52} which is approximately $2 \cdot 10^{36}$ possible end situations. In other words, brute-force computing all possible moves is too time consuming.

3 Gin Rummy learning methodology

In the previous Section we already saw that brute-force calculating every possible action is too time consuming and it requires massive computational power. In the AI (artificial intelligence) world, there are lots of techniques to deal with problems avoiding the use of brute-force. One of them is evolutionary learning. To create a competitive algorithm for Gin Rummy, this paper will use the evolutionary learning method to examine if it is suitable in solving Gin Rummy.

3.1 Evolutionary learning & cross-over

The idea is to use a neural network with co-evolution technique [2] to tackle the problem. Evolutionary learning is based on the idea of organic evolution, trying to let the algorithm evolve solutions to problems rather than trying to find fixed rules for them, by using the principles of evolution such as selection, mutation and recombination of populations. Starting with certain population(s), usually random, the program begins to learn. After each phase, the population is being changed, either by selection, mutation, cross-over etc. to form a new population. The algorithm continuously tries to find a population that performs better than the previous one.

The populations in this paper that are continuously evolving are the weights of the neural networks. With cross-over, random noise and Gaussian noise, the weights will be changed after each evaluation round. The player and the opponent each have their own neural network. At the beginning of the training, the player and opponent start using two random neural networks. The Artificial Neural Network (ANN) gives an output value; based on this value, the program decides whether to take a card from the drawing pile or the discard pile. After taking a card, a heuristic function will determine which card the player has to throw away to the discard pile. To improve the precision and the reliability of the neural network, experiments had to be done with hundreds of thousands of trainings. When one network is better, the other value of the other network will change slightly in the direction of the network with better performance. More on cross-over will be explained later in the section.

The neural network consists of 3 layers: an input layer, a hidden layer and an output layer. The input layer has 52 inputs and 1 bias node, thus 53 nodes in total (see Figure 2). Each input node represent a card of the game, except the bias node; input node 1 represent ♠ Ace, input node 2 represent

♠ 2 etc. The hidden layer has 26 nodes and 1 bias node. The output layer has only 1 node, which gives the final result of the neural network, deciding whether to draw from the discard pile or the draw pile (see Figure 2).

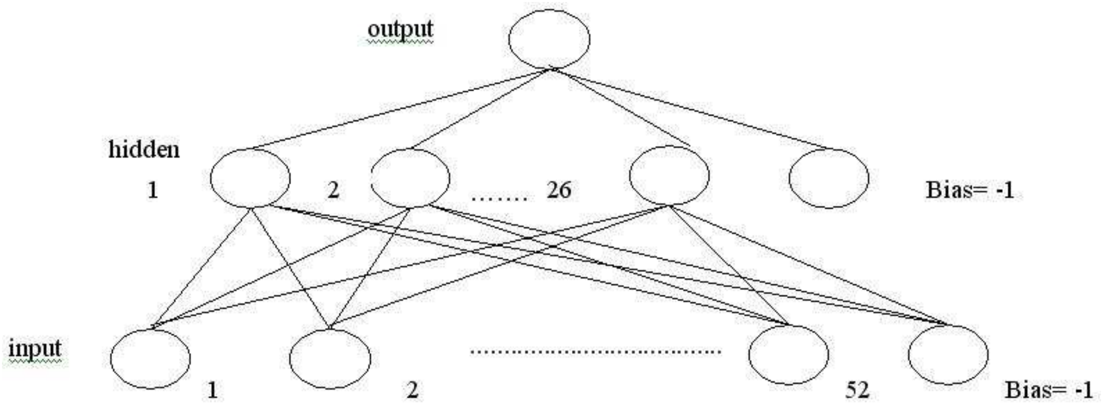

Figure 2: Artificial Neural Network (ANN)

The output node is also connected to all hidden nodes. To see if each input has the same chance, experiment are done using two different sets of begin weights for the input nodes. One set is to put all the begin weights between the nodes to 1. The other set contains random begin weights.

3.1.1 Cross-over

An epoch in the experiments consist of 2 or 3 pairs of games. When an epoch consist of 2 pairs of games, it actually means that the sequence of the cards of game 1 and game 2 are the same, and so is game 3 and game 4. The only difference is that in game 1 player 1 plays first, while in game 2 player 2 plays first. This mean that in game 2 for example, player 2 has exactly the same hand as player 1 had in game 1. The reason behind this arrangement is to eliminate the "luck" factor of the players. By playing with the same cards, we ensure that both players have the same chance to win the game. The results will than be based on the superiority of the weights of the neural network. After each epoch, the weights of the branches between the input nodes and the nodes in the hidden layer will be changed. Cross-over will occur depending on the number of games a player wins. In the experiment, we distinguish between single-direction cross-over and bi-direction cross-over.

Let say, we are now doing an experiment of 1000 epochs, where each epoch consist of 2 pairs of games using single-direction cross-over. This mean we will play 4000 games in total, and changes the weights at most 1000 times. After each epoch (4 games), the algorithm will evaluate if cross-over is needed to improve the weights of player 1. When player 2 wins more than half of the games, cross-over will occur. If the training is using an cross-over of 5%, then all the weights of the artificial neural network of player 1 will change 5% in the direction of player 2. This mean, the weights of the network of player 1 remains 95% the same and 5% of the weights of player 2 are added to the weights of network of player 1. See for example Figure 3. The weights of player 2 remains the same. The weights of the branches between the hidden layer and the output will be changed in the same way. In the single-direction cross-over, only player 1 is learning from the training. After the cross-over, the weights of player 2 will be changed by adding Gaussian noises.

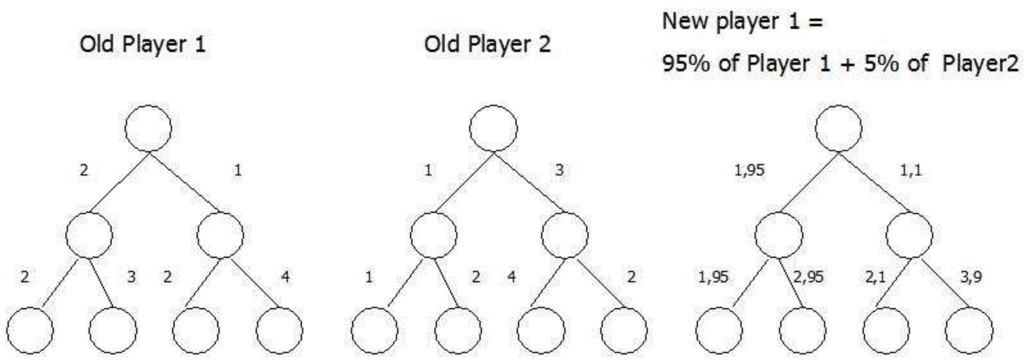

Figure 3: Cross-over

For the illustration we use a simplify network to show how cross-over in the experiments are being done.

When we are training with a cross-over of 10%, the neural network of player 1 will remain 90% the same and 10% of the weights of player 2 will be added to the weights of player 1. If player 1 wins more than half of the games or if both players have wins the equal amount of games, the weights of player 1 will not change since it is a single-directional cross-over. After every epoch, the weights player 2 will be changed by adding Gaussian noise, no matter which player wins more.

If the training is a bi-directional cross-over, the weights of the neural networks of both player can be changed. Depending on who wins most of the games in each epoch. In the bi-directional cross-over, both players learns from each other. If player 1 wins more than half of the games, the weights of player 2 will be crossed with 5% or 10% in the direction of player 1. The weights of player 1 remains unchanged. Vice versa when player 2 wins more than half of the games. Just like in the single-direction cross-over, Gaussian noise will be added to the weights at the ends of each epoch, irrespective of who wins.

3.2 Implementation of the networks (ANN)

3.2.1 The Artificial Neural Network

Each card has from the point of view of the agent 5 possible different states: IPH, IOH, IDP, TDP and UNK. The input of the network depends on the state of the cards. If it is in players hand (IPH) the input is 2, -2 in opponent hands (IOH), -1 if it is in the discard pile (IDP), 1 for the card just thrown away and 0 for unknown (UNK). Since the card just thrown away is not the same as the rest of the cards in the discard pile, it makes sense to give them different values. (A player can only take the top of the discard pile, the rest of the cards below can not be used). At first, the decision was made to evaluate the rest of the discard pile to -50, because these cards could not be reached anymore until reshuffling of the discard pile. However, it seems that -50 has too much impact on the neural network. Of course, the network can make adjustments, but it will need too much time to adjust and balance the network. Instead of -50, value 1 is chosen, which seems to be working quite well.

The value of the hidden nodes are being calculated using the following formula:

$$in_i = \sum_j W_{j,i} a_j,$$

where $W_{j,i}$ is the weight of input node j to hidden node i and $a_i = \sigma(in_i)$ the activation function.

The Artificial Neural Network (ANN) uses a sigmoid function σ to compute the activation function:
The formula of the sigmoid function is:

$$\sigma(\mathrm{x}) = 1/(1+e^{-\beta x}), \text{ with } \beta = 1.$$

Figure 5 gives an overview about the working of a neuron, with g as sigmoid.

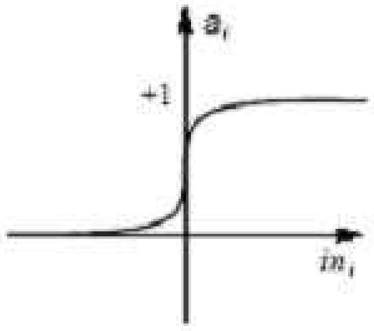

Figure 4: Sigmoid activation function σ. [11]

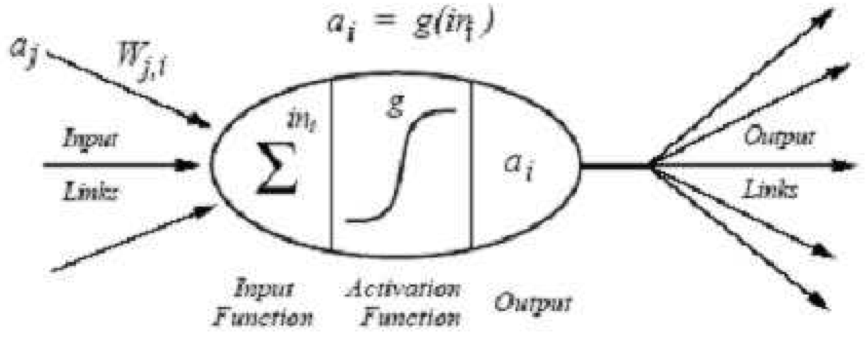

Figure 5: Neuron of the network [11].

3.2.2 The heuristic function

Once the decision is made which card to take, the algorithm needs to calculate which of the eleven cards to throw to the discard pile. The heuristic function calculates the card that is the best to throw at the discard pile. The function does this by judging the players hand, the cards in the discard pile and the known cards that the opponent has in his/her hand (the cards that the opponent takes from the discard pile). The idea is to make a heuristic function using a combination of minimal deadwood and chance rates to calculate which card is the most strategic card to be thrown away.

As described in the previous Sections, deadwood points consist of the face value of the cards that do not form sets. For example:
Case 1: $\heartsuit$ 5, $\heartsuit$ 6, $\diamondsuit$ 9 = 20 deadwood points.

Case 2: ♡ 5, ♢ 6, ♡ 9 = 20 deadwood points.
Although these two cases have the same number of deadwood points, it is clear that the first combination is better, because it almost forms a set. With a ♡ 4 or ♡ 7 it becomes a set. In case 2, it is just some "random" cards. So, the decision is made to count ♡ 5 and ♡ 6 as half as original. This means the number of deadwood points of the first case is 15 deadwood points ($\frac{1}{2} \cdot 5 + \frac{1}{2} \cdot 6 + 9 = 15$ rounded) and the second case stays the same as before, 20 deadwood points. The same with the following two cases:
Case 3: ♠ 5, ♡ 5, ♢ 8, ♢ 3 = 16 deadwood points.
Case 4: ♠ 5, ♢ 4, ♢ 9, ♡ 3 = 21 deadwood points.

An usual Gin Rummy game has about 20 hands, but when training the agent, a lot more "hands" are needed. To solve this problem, the cards in the discard pile are reshuffled again if the drawing pile runs out of cards. Each game has a maximum of 10,000 hands.Each epoch consists of 2 pairs of games. In each pair of games, the cards order is identical. In the first game, player 1 plays first and in the second game player 2 plays first. After playing 2 pairs of games, the algorithm checks which player wins more than half of the games. If player 1 wins more than 2 of the 4 games, the weights of player 2's network will be crossed 5% into the direction of player 1 and vice versa. For example when player 1 wins more than 2 of the 4 games, all the weights of player 2 stay 95% the same and take 5% of the weights between each node of player 1 and add it up to the corresponding branch of the network of player 2. In later experiments 6 games will be used instead of 4, and with another cross-over percentage, for example 10%.

3.3 Problems & solutions during implementation

One of the challenges was finding the "right" output value for the artificial neural network. To have an estimate of the output value, the neural network was run 1,000,000 (one million) times. After the test, 0.5 is chosen as threshold value for the neural network.

In some cases, the neural network comes to a deadlock, for example, when both players have similar hands. Player 1 throws a card, player 2 picks it up, throws the same card again and player1 picks it up again etc. Such situations can go on forever. In the experiment it happened that the player is doing the same actions more than 200,000 times. To solve this problem, random noise is added to the weights in addition to the crossover. Experiments have been done with random noise in the range between -0.01 and 0.01, and the range between -0.0001 and 0.0001. Even after adding noise to the weights, it some-times still required the network to do more than 100,000 hands to change the weights and escape this semi-deadlock. Therefore a maximum of 100,000 hands has been chosen as a limit.

Instead of random noise, tests were also conducted with Gaussian noise to find out if there are significant differences. These are generated using the Gaussian distribution:

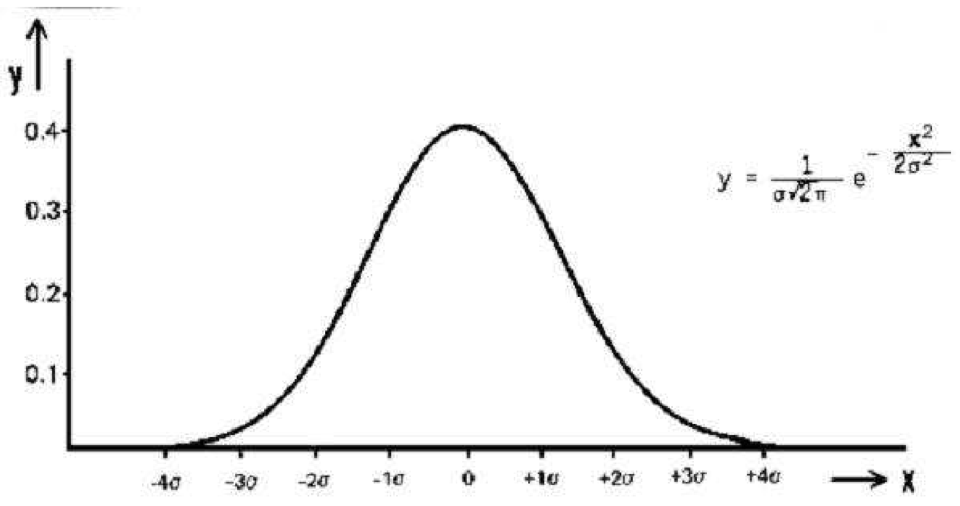

Figure 6: Gaussian distribution

Figure 6 is a Gaussian distribution showing the probability y of finding a deviation x from the mean ($x = 0$)

$$y = (1/\sigma\sqrt{2\pi})\ \cdot e^{-x^2/2}$$

where e is the base of natural logarithms and σ is the standard deviation.

In the experiments with bi-direction cross-over, both neural networks learn from each other and change their weights using cross-over. The experiments were done using Intel Pentium PCs with 1.7GHZ processors. Artificial neural networks using bi-direction cross-over were trained up to 500,000 games in approximately 8 hours. To test all the possible players (cross-over 5% with 4 games adding random noise, cross-over 5% with 4 games adding Gaussian noise, cross-over 10% with 4 games adding random noise etc.), several computers of LIACS (Leiden Institute of Advanced Science) were used to train the agents/players.

However the computers were not always available and even if they were available, it was not possible to use them for several hours straight. To save time and to utilize the available resources, the weights are being saved after every one thousand games of training. When the training needs to be resumed, the program just reads the weight being saved at the previous training.

Besides testing with Bi-direction with a maximum of 10,000 hands, experiments are also conducted with single-direction cross-over with a maximum 5,000 hands to see if it results in better performance. Instead of letting both artificial neural networks to learn from each other, what will happen if only one neural network learns from the other? One neural network does the same as bi-direction cross-over, but the other just adds Gaussian noise. Because of time constraints, these agents using single-direction were only trained half as much as in the case of bi-direction cross-over. But amazingly the performance is almost as good as the bi-direction cross-over. The results are shown in Section 4.

4 Training results

After explaining about the working of the neural network and the heuristic function in Section 3, let us see how each agent performs. To test what range of random noise to add to the weights, the average hands per game were measured. First using a random noise between -0.01 and 0.01 (Figure 7), then using a random noise between -0.0001 and 0.0001 (Figure 8). The average hands per game using a bigger random noise interval are significantly less than when using smaller random noise interval. It seems to be ideal to use the random noise which leads to less average hands per game, however the weights of the neural were too fast and too much using a random noise between -0.01 and 0.01. The changes in random noise were much bigger than the changes of the neural network using cross-over. After 50,000 games, some of the weights were already above 130. Figure 7 below show the average hands/turns per game. The training was done using 100,000 games, 25,000 epochs with a random noise between -0.01 and 0.01.

Figure 7 below show what the average hands/turns per game. The training was done using 100,000 games, 25,000 epochs with a random noise between -0.01 and 0.01.

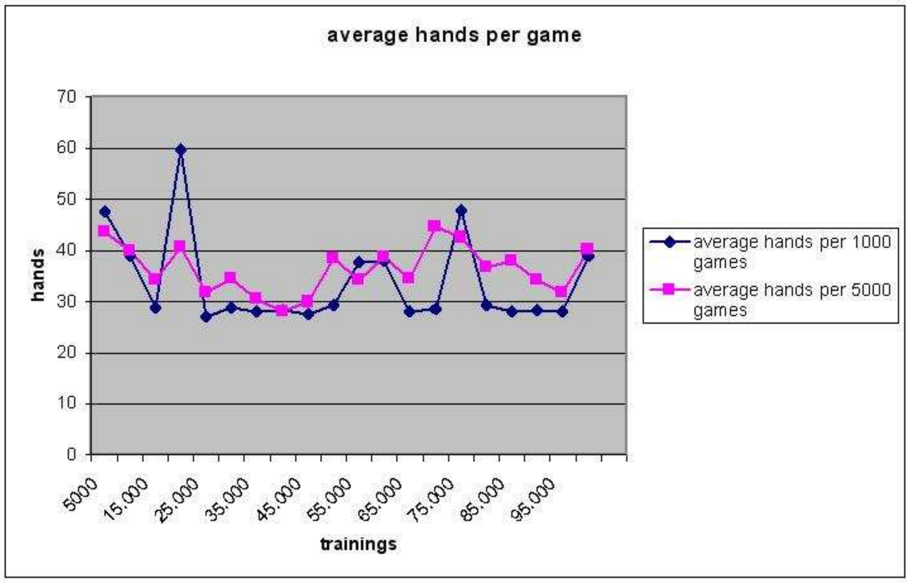

Figure 7: Average hands per game with random noise between -0.01 and 0.01

The number of average hands per game using random noise between -0.0001 and 0.0001 are much higher. This is because of the extra adjustment needed to change the weights. After each adjustment the weights are changed at most with 0.0001. Figure 8 gives the average hands per game with random noise between -0.0001 and 0.0001.

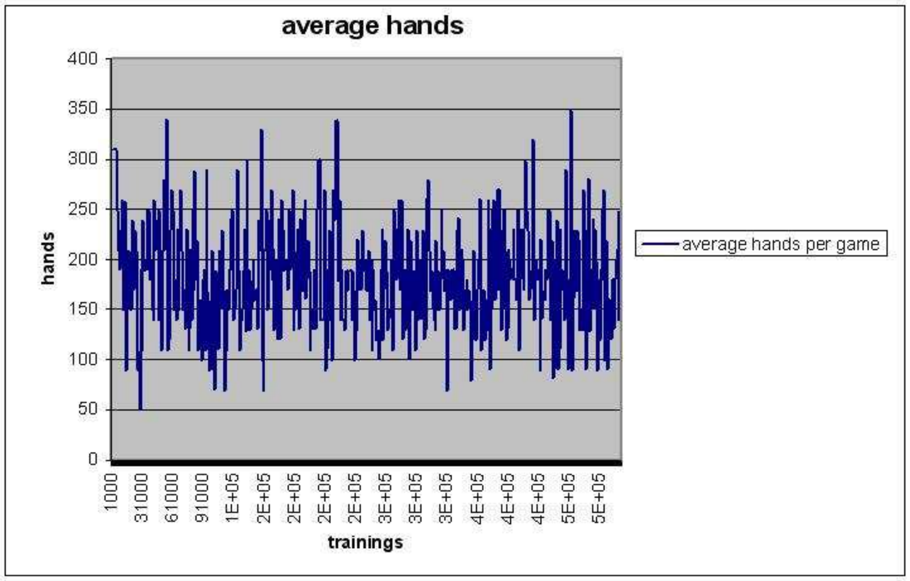

Figure 8: Average hands per game with random noise between -0.0001 and 0.0001

In the experiment with bigger random noise, the best result seems to be at a training sample of 45,000. The number of turns with bigger random noise is significantly smaller than the test with smaller random noise. After 500,000 trainings with random noise between -0.0001 and 0.0001, there is still no clear sign of decrease in the number of hands per game, perhaps more training is needed.

We now describe some experiments with different strategies. Rand1 is a random player that takes and throws away cards randomly. Using no strategy at all. Rand2 is similar to the other "players" using the same heuristic function to decide which card to throw, but does not have a Artificial Neural Network (ANN) to decide which cards to take. It just takes randomly from the discard pile and the draw pile. Below are 3 different tables. Each table show the results of a match between the different "players". All players plays against everybody, except itself. Table 1 shows the results of "players"

→ wins

↓ lose	Rand1	Rand2	Cross 5% 4 games	Cross 5% 6 games	Cross 10% 4 games	Cross 10% 6 games
Rand1	X	918	930	922	925	928
Rand2	0	X	606	637	617	621
Cross 5%, 4 games	0	385	X	505	498	494
Cross 5%, 6 games	0	348	495	X	499	496
Cross 10%, 4 games	0	377	495	494	X	497
Cross 10%, 6 games	0	372	503	499	499	X

Table 1: Standard begin-weight with random noise.

trained using standard begin-weight with random noise. Rand1 and Rand2 don't use any neural network at all. Rand1 is completely random and Rand2 takes random, but uses the same throw card function as the rest. Table 2 uses random begin weight with Gaussian noise and Table 3 standard begin-weight with Gaussian noise. All the players were trained 500,000 games with the neural network before entering the match.

The experiments done in Table 1, 2 and 3 all use bi-direction cross-over. To see if single-direction makes any difference, experiments were done using single-direction cross-over. Table 4 shows the result of the tournament of the agents/players using single-direction cross-over. Due to time constraints, the players using single-direction cross-over were only trained 250,000 times. However, it seems that the result are almost as good as using bi-direction cross-over. With fewer cross-overs, the neural network can train more games per hour making it learns "better".

As we can see from the tables, the methods we used to train the players does have influence on the outcome of the experiments. Table 2 and Table 3 both use the same Gaussian noise function, have the same players, but differ in begin-weights. Table 2 use random begin-weights, while Table 3 use standard begin-weights. When playing against player "Rand 2", the player trained with cross 5 % in 4 games perform the best in Table 2, while the player trained with cross 10 % 6 games does the better job in Table 3.

→ wins

↓ lose	Rand1	Rand2	Cross 5% 4 games	Cross 5% 6 games	Cross 10% 4 games	Cross 10% 6 games
Rand1	X	918	920	923	926	932
Rand2	0	X	602	588	588	581
Cross 5%, 4 games	0	384	X	504	499	495
Cross 5%, 6 games	0	405	490	X	498	516
Cross 10%, 4 games	0	405	498	498	X	503
Cross 10%, 6 games	0	404	502	481	493	X

Table 2: Random begin-weight with Gaussian noise.

→ wins

↓ lose	Rand1	Rand2	Cross 5% 4 games	Cross 5% 6 games	Cross 10% 4 games	Cross 10% 6 games
Rand1	X	918	929	911	928	913
Rand2	0	X	584	593	594	622
Cross 5%, 4 games	0	408	X	481	518	527
Cross 5%, 6 games	0	394	518	X	479	467
Cross 10%, 4 games	0	396	473	516	X	499
Cross 10%, 6 games	0	363	464	527	495	X

Table 3: Standard begin-weight with Gaussian noise.

→ wins

↓ lose	Rand1	Rand2	Cross 5% 4 games	Cross 5% 6 games	Cross 10% 4 games	Cross 10% 6 games
Rand1	X	918	966	962	964	972
Rand2	0	X	573	589	581	581
Cross 5%, 4 games	0	421	X	476	496	486
Cross 5%, 6 games	0	403	518	X	488	492
Cross 10%, 4 games	0	406	496	507	X	494
Cross 10%, 6 games	0	409	511	499	497	X

Table 4: Single-direction cross-over after 250,000 trainings.

Table 2 and Table 4 differ in the way cross-over is being done. Table 2 use bi-direction cros-sover (both players learn from each other), while Table 4 use a single-direction cross-over (only one player is learning). Judging from the results played with player "Rand 2", using bi-direction cross-over performs slightly better than single-direction cross-over in general.

Player "Rand 2" uses the same heuristic function to throw card, but does not have a neural network to compute where to pick the card from (discard pile or draw pile). When judging the performance of the different players, it is easier to benchmark them with player "Rand 2".

5 Discussion & Conclusion

5.1 Is ANN useful for solving Gin Rummy?

Judging from the result of the three tables, it seems that there are no major differences between using standard begin-weights (all weights are set to 1), and using random begin-weights, with Gaussian noise or with random noise. Comparing Table 2 and Table 3, there is no clear evidence which player performs better. In Table 2, it seems that the player trained with cross-over 5% in 4 games performs better than the rest when playing against Rand2. In Table 3, it is the player who is trained with cross-over 10% in 6 games that does the better job when playing against Rand2. This conclusion is arguable, since other players do better in other matches. In either way, the results show that using an Artificial Neural Network, does perform better than Rand2, a player that does the same as the other players, but that does not have a neural network and takes card randomly. Perhaps the neural network needs to be trained much longer, or perhaps the random noise and Gaussian noise are influencing the neural network too much.

Table 4 shows the result using single-direction cross-over. With half as much training as bi-direction, single-direction cross-over performs quite good. Using single-direction, a higher number of trainings per hour can be reached than with bi-direction cross-over. However, from the experiments and the results shown in all four tables, it does not really matter which player/agents you choose, no matter how small the adjustment after each evaluation round. When the Artificial Neural Network is trained long enough, the performance will increase. But the number of games training will most likely increase much faster than the performance.

We can indeed use Artificial Neural Network to solve Gin Rummy. As long as the neural networks are trained long enough, it does not matter if you choose bi-direction or single-direction, cross-over 5% or 10%, the performance will increase. However, choosing the right one, will save the network a lot of training time.

5.2 Further research

Due to time constraints, it was not possible to train and test every possible combination of the several parameters. For example, cross-over 15% instead of 5% and 10%, cross-over after 8 games played instead of 4 or 6 games. Further training of the current setup is needed to see which setup yields a better performance in shortest time. Another possibility is to Improve the heuristic function to throw cards. Tests can also be done using more than 27 nodes (26 hidden nodes + 1 bias node) in the hidden layer or add another hidden layer between the current hidden layer and the output layer. The neural network can also further be improved by playing with experts and using dataming [13] to adjust the weights of the Artificial Neural Network.

Beside of using an Artificial Neural Network to solved Gin Rummy, we can also use co-evolutionary learning to solve other (card) games like Poker, Bridge, Hearts, Checkers.

References

[1] Gerald Tesauro (2002), Programming backgammon using self-teaching neural nets. Artificial intelligence 134 (2002), 181–199.

[2] Jordan B. Pollack, Alan D. Blair & Mark Land (1996), Coevolution of a Backgammon Player. Proceedings of the fifth AI conference, May 1996, Nara, Japan.

[3] Kimon Tsinteris, David Wilson (2001), TD-learning, neural network, and backgammon.
http://www.cs.cornell.edu/boom/2001sp/Tsinteris/gammon.htm.

[4] Gerald Tesauro (1995), Temporal Difference Learning and TD-gammon. Communications of the ACM, vol. 38, no. 3.

[5] Jonathan Baxter, Andrew Tridgell & Lex Weaver (1998), TDLeaf(lambda): Combining Temporal Difference Learning with Game-Tree Search. Australian Journal of Intelligent Information Processing Systems, Vol. 5 No. 1 (1998), 39–43.

[6] Clfford Kotnik, Jugal Kalita (2003), The Significance of Temporal-Difference Learning in Self-Play Training, TD-rummy versus EVO-rummy. University of Colorado at Colorado Springs.

[7] Gerald Tesauro (1992), Practical Issues in Temporal Difference Learning. Machine Learning 8, 257–277.

[8] Stuart Russell, Peter Norvig (2003), Artificial Intelligence, A modern approach. Prentice Hall, second edition.

[9] Walter B. Gibson (1974), Hoyle's Modern Encyclopedia of Card Games. New York: Doubleday.

[10] Gin Rummy game history.
http://www.playjava.com/gin_rummy_game_history.html

[11] W. A. Kosters (2006), Kunstmatige Intelligentie college sheets, University Leiden, http://www.liacs.nl/~kosters/AI

[12] G. de Croon, M.F. van Dartel & E.O. Postma (2005), Evolutionary Learning Outperforms Reinforcement Learning on Non-Markovian Tasks. University Maastricht, The Netherlands.

[13] W. A. Kosters (2002), Dealing with the Data Flood: Mining Data, Text and Multimedia (J. Meij, editor), STT/Beweton, The Hague, Chapter 6.2.8, 641–645.