Designing an Intelligent Carrom Player

Abhijeet Dubey
16305R006

Nidhi SIngh
163059004

Rishabh Dabral
164050002

November 14, 2016

Contents

Abstract

This report analyses the performances of several agents designed to play the game of Carrom in single player and double player mode. We propose several possible agents and highlight the challenges accompanying them at the time of implementation. Our best agent clears the board in 23 turns on average in single player mode. The agent follows a deterministic policy arrived upon by several evaluations. We also propose a different state representation for the agent in 2 player mode.

Chapter 1

Introduction

This piece of work involves designing an agent that is capable of playing Carrom in two modes. In 1-player mode, the agent is expected to clear the board in as less moves as possible. In 2-player mode, the agent has to pocket all the coins of the colour assigned to it before the opponent does the same while also keeping in mind that the queen must be pocketed by either of the agents before clearing the board.

The game is simulated using a client-server model wherein the agent acts as a client which queries the server (or the environment) with an action and receives a reward and the next state for the corresponding action. The action played by the agent is represented by a 3-tuple containing, (a) the striker's position on the strike line in the range 0 to 1, (b) the angle of strike from the striker's position in the range -45 to 225 degrees, and, (c) the force with which the striker has to be hit, again in the range 0 to 1. All the elements of the action space are continuous, thus forming a three dimension continuous action space.

Not surprisingly, Carrom shares some features with other games on which some research has already been done. For example, it shares the property of being a stochastic game with Billiards-like pool games. Additionally, both are continuous state space based board games and both can be considered recursive as well as extensive form games. Further, if played in 2-player mode, both have turn-taking structure and. Finally, the next space depends on the accuracy with which an action is actually executed, thus, making 'noise' a significant factor in both games.

Carrom can also be considered similar to Soccer as both the games have continuous action and state spaces and share the property of stochasticity. But at the same time Soccer is somewhat different since it features continuous control and concurrent actions by both teams.

The game also inherently posses certain peculiarities that need to be accounted for while designing our agent. For example, since the state space is continuous, we can not store the states and their values. The same is the case with action space. Random noise is also added to the action sent by agent which leads to non-deterministic outcomes. It was difficult to train the model since we did not have any expert dataset to train our model on.

Chapter 2

Proposals

In this chapter, we enlist some proposals action strategies and the corresponding state representations. Our approaches can be divided in three broad categories, namely - Continuous Actor-Critic, Trial and Error and Policy Search.

2.1 Continuous Actor-Critic Method

The fact that the action space is continuous lead to contemplating on methods that are designed to learn in this setting. We attempted to use the Actor-Critic method as described by Sutton and Barto in their book. The approach involves designing a function approximator for state evaluation a neural network for representing the policy (also know as a policy network). This approach is somewhat similar to the one used for the agent that played AlphaGo. We shall now discuss the components of the model one by one.

2.1.1 State Representation

For this model, we represent the state by dividing the board into 32x32 grids. We create four such tilings with different offsets. Each grid in each tile is given a value of '1' if it contains a coin and '0' if contains no coin. Thus, our state is a 4x32x32 3-dimensional binary array.

2.1.2 State Value Approximator

Considering the tile coding as described earlier, the State Value function can be best represented by a Convolutional Neural Network (CNN). The input to this CNN would be the 3 dimensional state array and the output would be a single neuron representing the state-value function of the given state. Typically, we tried a CNN with 3 convolutional layers with 2x2 maxpooling layers which were subsequently fed to a fully connected feed-forward neural network with one hidden layer. The update rule the network weights was derived by the TD error:

$$cost = (R + \gamma V(s') - V(s))^2$$

The backpropagation algorithm is asked to reduce this cost through any appropriate optimization algorithm.

2.1.3 Policy Network

The policy network took the state as its input and three neurons, representing the three action components, as the output. This network was architecturally similar to the neural network for state value approximation. The update to this network is based on the TD update experienced by the state value approximator.

In essence, the error propagated backwards to the network ought to be inversely proportional to the TD error. However, when the state value function approaches optimality, the TD error is bound to reduce. As such, reporting a high error would mislead the policy network. To take care of this fact, we introduce a decaying parameter β that we multiply with the error. The loss can then be represented as:

$$cost = \beta \exp^{-\delta}$$
$$\text{where, } \delta = R + \gamma V(s') - V(s)$$

β is taken as a 3-dimensional vector corresponding to each output.

2.1.4 Training

Training such a network requires running several episodes of single player game. To speed up the learning process, we tried to first make the agent learn policy of the agent that has been chosen using the trial-and-error approach as discussed in section 2.2. This was expected to tune set the weights such that the neural network learn to output the actions in the desired range.

Figure 2.1 demonstrates the flowgraph to be followed for training such an agent. The agent first receives the state and the reward of the previous state from the environment. It then passes the state and reward to the state-value function approximator which uses it to update its weights as per the update rule mentioned above. Note that the current state is treated as *V(s')* by the approximator. The approximator then returns the TD error, δ, to the agent which is then forwarded to the policy network.The policy network uses the β and δ values to update its weights and returns the action for the current state back to the agent. The agent finally sends the same action to the environment and this cycle goes on for several episodes.

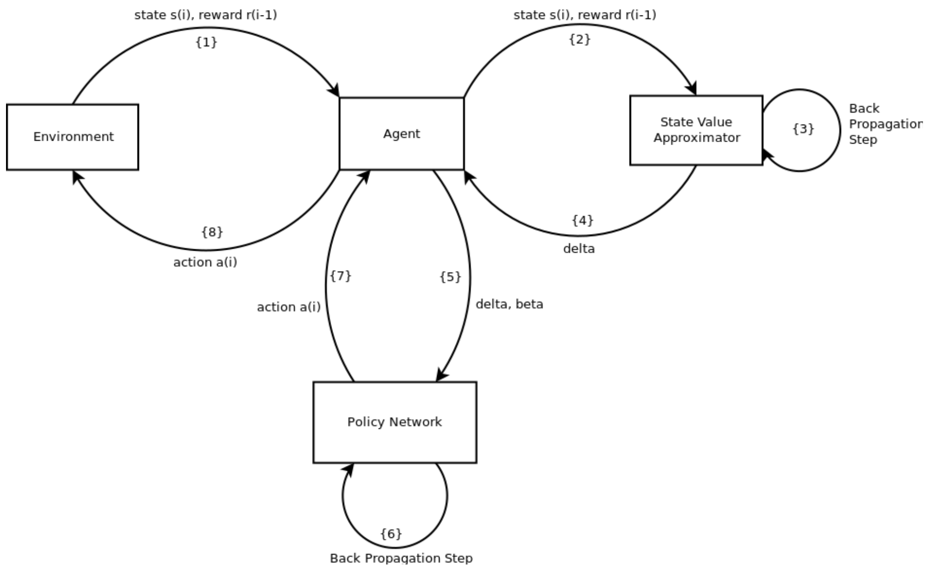

Figure 2.1: Flow Graph for training continuous actor-critic agent.

However, a major challenge in training such a network was posed by the random moves that the server played on receiving an infeasible action. Unfortunately, the policy network did not convincingly converge to the values within the stipulated ranges of striker position, angle and force.

2.2 Trial and Error based approaches

Since the game of Carrom is supported by a strong domain knowledge, using them to design an agent that performs under the set thresholds was considered. Although the choice of action across various states of the game was, in part, dictated by the rules of the game, yet coming up with a suitable state representation went a long way in improving the performance. In such attempts, we tried to analyse grid-based state representation of the board.

2.2.1 Generic grid based state representation

The basic idea behind grid-based state representation was to divide the board into certain areas (grids) and consider the number of coins in grids as state. For example, if we divide the board into 64 grids, the state would be a 64-tuple wherein each element would store the number of coins in the corresponding grid (see Figure 2.2. Algorithm 1 describes a generic algorithm that might be used to take actions based on the grid-based representation. A point worth noting in this approach is that we do not try to hit the coin to the pocket but instead to just hit the coin somehow. One major reason that lead to this choice was the noise as any effort to pocket the coin was nullified by the noise.

```
   Input: State $s$ which consists of location of coins
   Output: Action $a$
1  GenericGridBasedAgent($s$):
2  Divide the carrom board into n grids
3  max_grid ⟵ grid with maximum number of coins
4  coin ⟵ pick a coin uniformly at random from max_grid
5  Take action such that the striker hits that coin
```

Algorithm 1: A Generic Grid Based Agent

In our proposed model, we chose to divide the board into six grids. The division of the board is depicted in Figure 2.3. This division was only natural as these locations required different sets of actions. So, say, if the grid 2 had a high density of coins, our agent would attack one of the coins in grid 2 from the position 0.0 and force depending on the number of coins on the table.

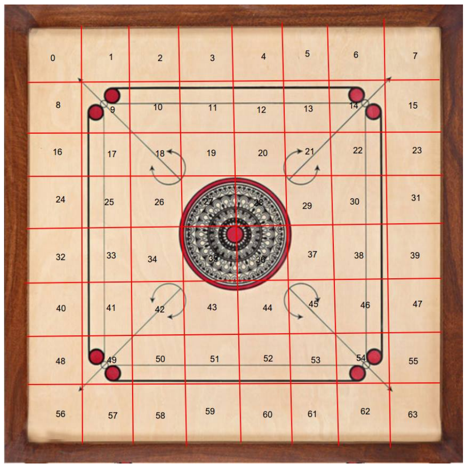

Figure 2.2: Grid Structure 1.

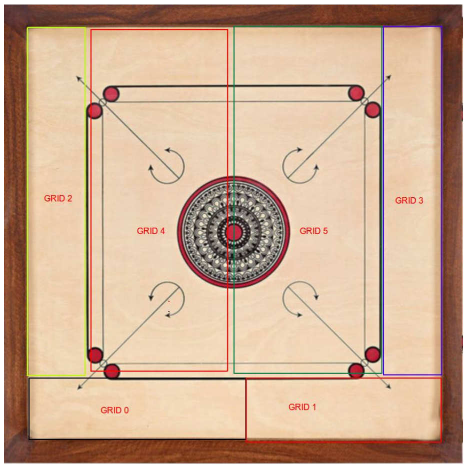

Figure 2.3: Grid Structure 2.

2.2.2 Proposed 1 Player Agent

We now discuss the policy that our agent follows. Algorithm 2 describes the strategy that we propose. The agent first attempts to hit the coins lying below the striker-line. For this, she compares the density of the two grids lying below the line and targets a coin in the grid with higher density. If no coin in grids

0 and 1, she follows the same comparison-based approach with grids 2 and 3, viz. the left and right columns beyond the strike-line, respectively. The striker position is set to the closer of the two tips of the strike-line. Finally, the agent focuses on the two middle grids. If one of the two grids contains the queen, the agent attacks the queen. Otherwise, she attacks any random coin lying in the denser of the two grids from the middle position on the strike-line.

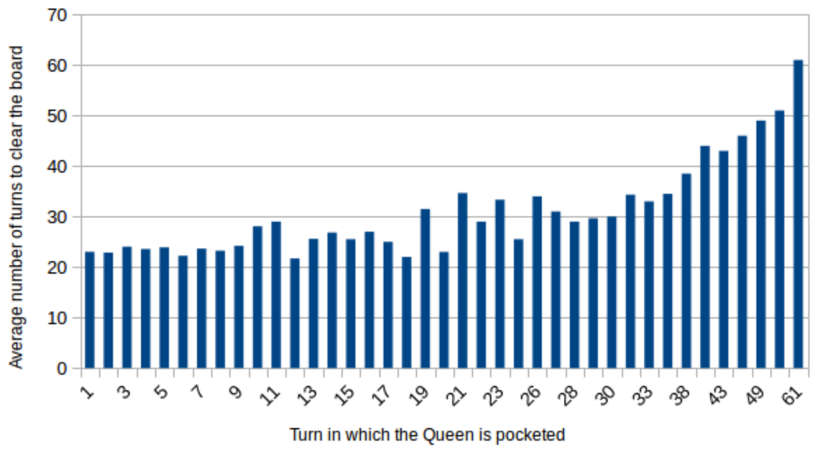

Figure 2.4: Plot between queen pocketed turn and average no. of turns

The above policy performed better than other tweaks that we attempted. In addition to trying other grid breakups, we also tweaked with the force value for different number of coins on the board. Additionally, we also tried alternate orderings of grids to attack. Finally, we also attempted to set our agent in a way that keeps on targeting the queen unless it is pocketed. However, this tweak also degraded the performance. Finally the policy that outperformed other alternatives was the one which is targeting the queen neither too early nor too late. This is a positive aspect as it limits the effect of the queen in the game. Figure 2.4 illustrates the effect of the time at which the queen is pocketed on the total number of turns it takes to clear the board. Clearly, there is no visible pattern in the plot for the initial 20 turns. The bars begin to get taller after 20 as they, most likely, refer to the case when the queen has stayed till the end. This illustrates the fact that the queen should not stay till the end. Our model takes into account this observation and our strategy is to pocket the queen neither too early in the game nor too late.

```
Input: State $s$ which consists of location of coins
Output: Action $a$ which is a three dimensional vector
SinglePlayerAgent($s$):
/* getCoins will return location of coins and queen */
coins, queen = getCoins($s$)
freq ⟵ $frequency\ of\ coins\ in\ 6\ grids$
for each coin in coins: do
    location ⟵ $grid\_number\_of\_coin$
    /* Target coins in grid 0 and grid 1 */
    if location = grid 0 and freq[grid 0] >freq[grid 1] then
        calculate angle
        position ⟵ 0.75
        force ⟵ 1
        action ⟵ ($position, angle, force$)
    end
    else if location = grid 1 and freq[grid 0] <freq[grid 1] then
        calculate angle
        position ⟵ 0.25
        force ⟵ 1
        action ⟵ ($position, angle, force$)
    end
    /* Target coins in grid 2 and grid 3 */
    else if location = grid 2 and freq[grid 2] >freq[grid 3] then
        calculate angle
        position ⟵ 0
        force ⟵ 1
        action ⟵ ($position, angle, force$)
    end
    else if location = grid 3 and freq[grid 2] <freq[grid 3] then
        calculate angle
        position ⟵ 1
        force ⟵ 1
        action ⟵ ($position, angle, force$)
    end
    /* Target queen or coins in middle grids */
    else
        If queen is present then target queen by placing striker in the
          middle otherwise target coins in middle two grids.
    end
end
```

Algorithm 2: 1 Player Agent

In our discussion so far, we have tacitly assumed the force to be optimal. However, experiments with different force values could not improve our results. The dynamics of force will be discussed in section 3.2.1.

2.2.3 Proposed 2 Player Agent

For the 2-player version of the game, we slightly tweak the 1-player agent. Algorithm 3 describes the policy used by the agent that we submitted. Instead of considering all the coins on the board, the agent focuses only on the frequency of the coins of her colour. The rules of the game are set in such a way that the queen must be pocketed before the game ends. To make sure that she doesn't end up clearing the board before pocketing the queen, the agent begins to target the queen as soon as only 5 coins of the her colour remain on the board. Before this and after the queen is pocketed, the strategy remains the same as 1-player agent.

```
   Input: State $s$ which consists of location of coins
   Output: Optimal Action
 1 TwoPlayerAgent($s$):
 2 /* getColoredCoins will return location of white coins, black coins and
    queen */
 3 whiteCoins, blackCoins, queen = getColoredCoins($s$)
 4 white_freq ⟵ frequency of white coins in 6 grids
 5 black_freq ⟵ frequency of black coins in 6 grids
 6 if color = “White” then
 7     if number of white coins are less than 5 and queen is not pocketed
        then
 8         Greedily target queen before targeting white coins.
 9     end
10     else
11         Use the same same strategy as 1 Player Agent but this time target
            only white coins instead of targeting all the coins.
12     end
13 end
14 else if color = “Black” then
15     if number of black coins are less than 5 and queen is not pocketed
        then
16         Greedily target queen before targeting black coins.
17     end
18     else
19         Use the same same strategy as 1 Player Agent but this time target
            only black coins instead of targeting all the coins.
20     end
21 end
```

Algorithm 3: 2 Player Agent

Chapter 3

Performance & Analysis

The previous chapter discussed in detail some of the possible strategies to implement the agent. We now compare the performances of those strategies upon implementation. We propose two metrics for evaluation of an agent; the number of turns taken to clear the board after 500 experiments and the plot of average score per turn.

3.1 Experiments with Grid sizes

One obvious design parameter in grid-frequency based state representation is the way we segment the board. We experimented with three grid designs - A 8x8 grid, A 6x6 grid and a special grid division (see Figure 2.3). Table 3.1tabulates the best performances observed on these designs. Clearly, the special grid division outperformed the other two generic grid divisions.

3.2 Tweaking the Special 6-Grid division

Based on the above results, it was clear that the proposed 6-Grid division of the board yielded better results. We then tried several tweaks on this design. Two most important aspects behind getting the agent to function well were the

Grid Type	Average Number of Turns
6x6 Uniform Grid	28
8x8 Uniform Grid	31
Special 6-Grid division	23

Table 3.1: Comparison of proposed grid designs

values of power and the algorithm to select the action.

3.2.1 Deciding the Force value

The stipulated rules of the game do not penalize the agent with extra coins if the striker is pocketed. At most, the coins pocketed in the current hit get replaced if the striker is pocketed. This motivated us to hit the striker with full force every time. This decision was further cemented when we observed that choosing a dynamic force selection policy resulted in degradation of performance on almost all occasions in the long run. Figure 3.1 plots the average score received for every turn across 500 experiments. As can be seen, the simple, full-force, policy outperformed the other complicated dynamic force selection policies.

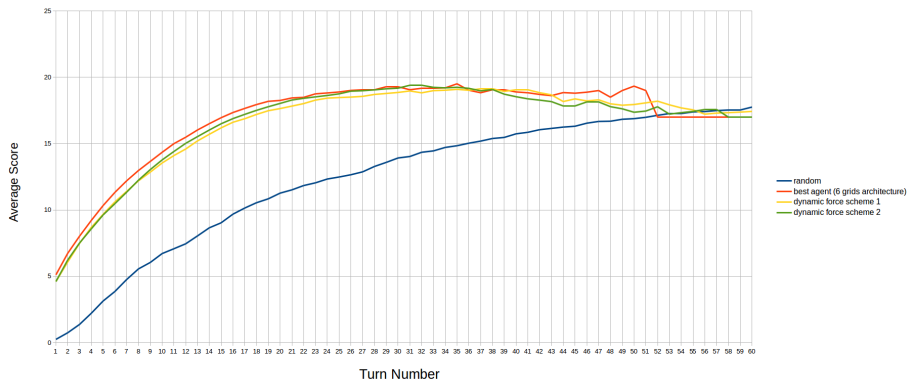

Figure 3.1: Performance plots for different strategies.

3.2.2 Choosing the best algorithm

In our discussion, we refer to the term ’algorithm’ as a choice of relative ordering of the six grids in which they should be attack. Three different algorithms were considered and the best was chosen for the final agent. Table 3.2 contains their relative performances. In the table, we represent an algorithm by the order in which the grids are attacked.

3.3 Performance of Continuous Actor-Critic Method

As was mentioned in Section 2.1.4, training a Continuous Actor-Critic method based agent proved to be a difficult task. It’s performance could only result in

Relative Ordering of Grids	Average Number of Turns
0/1-2/3-4/5	23
2/3-0/1-4/5	29
4/5-0/1-2/3	26

Table 3.2: Comparison of three algorithms for the Special 6-grid division

a slightly better performance as compared to the random-action playing agent which clears the board in roughly 120 hits. This is expected as more often than not, the output of the policy network turned out to be an invalid action. We shall discuss the possible reasons for its failure in section 3.4.1

3.4 Analysis

It is evident from the discussion above that the best model uses a deterministic, rule-based policy which has been arrived upon after several hits and trials. The fact that a learning-based agent could not perform well for us does not signify that they can not be the agent of choice.

3.4.1 Why Learning Failed?

We suspect the flaw in our learning process was in the way we updated the weights of the Policy Network. A policy network inherently requires supervision from an expert, just the way the agent for AlphaGo was trained. We clearly lacked this supervision. However, we did try to force the agent to learn the deterministic policy used by our other agents and expected it to learn the limits in which the action values need to be output. These attempts could not see fruition due to fact that the environment plays a random action on encountering an invalid angle. Thus, the state-value approximator did not recieve an accurate TD error which, in turn, affected the backpropagation of error in the policy network. Not surprisingly, the actions continued to be invalid.

3.4.2 Fouls are, apparently, harmless!

While choosing the best deterministic policy, we sought the correlation between the number of fouls and the total number of turns taken to clear the board. As it turned out, trying to reduce the number of fouls in turn affected the rate at which the coins were being pocketed. That was because as we reduced the power of hits, several coins - that would have otherwise been pocketed after some rebounds - did not reach the pocket. Consequently, we decided to stick to the policy of hitting with the full force all the time.

For two player agent, we have, however, chosen to reduce the force when less than five coins of our colour remain on the board. This is expected to reduce the risk of mistakenly pocketing the queen when only one coin is left in the end. Additionally, this would prevent pocketing unwanted coins of the opponent's colour.

Clearly, fouls are harmless for 1-player agent, but not quite so for a 2-player agent.

Chapter 4

Conclusion

In this report, we presented a description of our experiments, observations and analyses in the endeavour to design an agent that is capable of playing the game of Carrom in 1-player and 2-player modes. The game, being a continuous state and continuous action space game, exhibited similarities to other board based games like Billiards. We proposed two broad approaches towards designing the agent. In our first approach, the Continuous Actor-Critic based approach, we relied on designing two CNN's for representing a state-value approximator and a policy network. In our second approach, we contemplated upon segmenting the board into grids and chosing an action based on the density of coins in the grids. Next, we evaluated the performances of our proposed agents and progressively establish the supremacy of the chosen agent over the other proposed agents. Towards the end of the report, we also analysed the reasons behind the failure of some of our approaches and discussed how other peculiarities of the game dictated our choice of the agent.

Our best agent in 1-player mode clears the board in 23 turns on average. Although the 2-player agent has outperformed all the other agents that we came across, the performance of the 2-player agent is yet to be tested.

Bibliography

[1] Christopher Archibald and Yoav Shoham. Modeling billiards games. In *AAMAS*, pages 193–199, 2009.

[2] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[3] Chris J. Maddison Arthur Guez Laurent Sifre George van den Driessche David Silver, Aja Huang. Mastering the game of go with deep neural networks and tree search. 2016.