Building an agent to play the game of Carrom

Team Rocket, Final Project, CS747, Autumn 2016

Abstract

We had looked at different ways of designing and implementing an agent to play the popular game of Carom in both single player mode and double player mode. In this report, we summarize our observations and the obtained results in designing different agents.

1 Introduction

One way of representing a state s in game of carrom is maintaining a list of positions of all coins and their color. Any action a can be represented as an ordered triplet of position of striker, angle with base line and force with which the striker is released. A possible strategy for developing an agent to play this game is, for any given state s, we calculate $Q(s, a)$ for every possible action a, and take the action which corresponds to maximum Q-value. But implementing this is not as easy as it seems, because number of states and actions are not finite. So, in this report we have described our approaches to find a feasible way of implementing such an agent. This report is organized as follows: In second section, we discuss about the usefulness of Q-learning in deciding the optimal action and necessity of neural networks to predict Q-values in spite of having a simple algorithm to compute them. In third section, we discuss about three different agents which we have developed in process of finding the optimal one. In fourth section, we provide our results for the best agent we have developed and in the last section, we have summarized our insights and observations regarding these agents and suggested some possible improvements.

2 Q-Learning

2.1 Simple algorithm to compute Q-values

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989). Its simplest form, one-step Q-learning, is defined by following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Algorithm 1 Q-learning algorithm

procedure COMPUTE Q(S,A)
    Initialize $Q(s, a)$ arbitrarily
    repeat(for each episode):
        Initialize s
        repeat(for each step of episode):
            Choose a form s using policy derived from Q (e.g., ϵ-greedy)
            Take action$a$, observe r, s'
            $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_a Q(s', a') - Q(s, a)]$
        until s is terminal
    until No more episodes are available
end procedure

So, we could have a table that stores the Q-value for every possible state-action pair and iteratively update this table as we play games. Our policy π would be based on choosing the action with the highest Q value for that given state.

2.2 Motivation behind using Neural Network

Our state-action space is too large to store in a table. Imagine a particular state of carrom on a board. In a tabular method, if the position of a single coin changes by just a single pixel (or block), we have to store that as a completely separate entry in the table. Obviously that's silly and wasteful. What we need is some way to generalize and pattern match between states. We need our algorithm to say "the value of these kind of states is X" rather than "the value of this exact, super specific state is X". This is the reason, why we want to use neural network to predict Q-values. In neural networks, a small change in input only brings a small change in output. So, even if there is a small change in state, Q-values would not change which is anticipated. We have used **PyBrain** library to create and train a neural network.

3 Algorithms for different agents

For the first agent, we have quantized action triplets to find best action at a given stage of game. In second agent, we have tried to predict the coin which should be targeted and decided the action based on the position of the selected coin. In the third one, we tried to predict the force, followed by a deterministic way of selecting coin, position of striker and the angle at which it should be released.

While training these agents, reward for pocketing striker is given as -2. It was done so, to differentiate those actions from the ones which result in pocketing no coins. Otherwise, agent may pick an action without considering the negative effect foul. The actions which do not pocket any coin are discouraged by awarding -1. Reward in other case is the score difference while for pocketing queen it is +3.

3.1 Learning Algorithm-I

3.1.1 Motivation

We want to find $Q(s, a)$ for all the actions given state s as input. Then we choose an action corresponding to the highest Q-value. As discussed earlier, it is not possible to implement this using simple algorithm given in section 2, due to infinite state-action space. So, we build a neural network to predict $Q(s, a)$ for all the actions given a state s.

3.1.2 Approach

Firstly, we train the network for 500 games and update the weights of the network after every action. And then use the network to predict $Q(s, a)$ while playing the game and based on those values, action is chosen.

Algorithm 2 Algorithm to train neural network

procedure TRAINING
  Initialize Q network arbitrarily
  function EVALUATE(s)
    Input: state s
    Output: vector t, where $t[i] = Q(s, i)$, $i \in A$
    Run Q network forward with state s as input
    Store output of Q network in t
    return t
  end function
  function TRAIN(s, $target$)
    Input: state s, vector $target$
    Output: Network Q
    $output \leftarrow$ Evaluate(s)
    Update network Q based on Error($output$, $target$)
    return Q
  end function
  repeat(for each episode):
    $s \leftarrow$ initial state
    $target \leftarrow$ Evaluate(s)
    repeat(for each step of episode):
      Choose a as $argmax_i\, target[i]$ following ϵ-greedy policy
      Take action a, observe reward r and new state s'
      $t \leftarrow$Evaluate(s')
      $maxQ \leftarrow argmax_i\, t[i]$
      $target[a] \leftarrow r + \gamma \cdot maxQ$ where γ is discount factor
      Train(s, $target$)
    until s is terminal
  until No more episodes are available
end procedure

3.2 Learning Algorithm-II

3.2.1 Motivation

In the previous algorithm, the optimal action *(position,angle,force)* for the given state predicted by the neural network was not showing good performance. So, we decided to change the output of the neural network.

3.2.2 Approach

In this approach, rather than training the neural network to decide the optimal action, we are training it to decide the optimal coin to hit and use a deterministic algorithm to hit that coin into pocket. Here set C is the set of coins left on the board at any instant and hit_coin is the target coin.

3.2.3 Algorithm

Algorithm 3 Algorithm to train neural network

procedure TRAINING
  Initialize Q network arbitrarily
  function EVALUATE(s)
   Input: state s
   Output: vector t, where $t[i] = Q(s, i)$, $i \in C$
    Run Q network forward with state s as input
    Store output of Q network in t
    return t
  end function
  function TRAIN(s, $target$)
   Input: state s, vector $target$
   Output: Network Q
    $output \leftarrow$ Evaluate(s)
    Update network Q based on Error($output$, $target$)
    return Q
  end function
  repeat(for each episode):
    $s \leftarrow$ initial state
    $target \leftarrow$ Evaluate(s)
    repeat(for each step of episode):
      Choose hit_coin as $argmax_i\, target[i]$ following ϵ-greedy policy
      Decide action a to hit hit_coin, observe reward r and new state s'
      $maxQ \leftarrow argmax_i$ Evaluate(s')
      $target[hit\_coin] \leftarrow r + \gamma \cdot maxQ$ where γ is discount factor
      Train(s, $target$)
    until s is terminal
  until No more episodes are available
end procedure

3.2.4 Variation in learning method (Batch Mode)

As a variation of above algorithm, we tried batch mode of operation for training the neural network. In this mode, rather than training the network immediately after every move against the target vector, we maintain a buffer of all the target vectors along with inputs and train only after the game is completed.

3.3 Learning Algorithm-III

In this case, we tried to predict the force, using a deterministic approach of selecting coin, position of striker and the angle at which it should be released. Intuition behind this approach is developed from observing the way our previous agents have played. Striker is being pocketed more often than expected in case of previous agent. So, to control the force, we decided to train the network to predict the force, hoping that striker does not follow the coin to the pocket.

3.4 Deterministic Algorithm

We implemented a procedure to pocket a given coin at any location on board in a noise-less environment. Our program takes position of coin as input and decides to take an action which results in pocketing that particular coin. Using that algorithm, agent takes just 12 moves to clear the board in noise-less environment in single player mode giving priority to queen as long as it stays in the game.

4 Results

For the agent which we have submitted for evaluation:

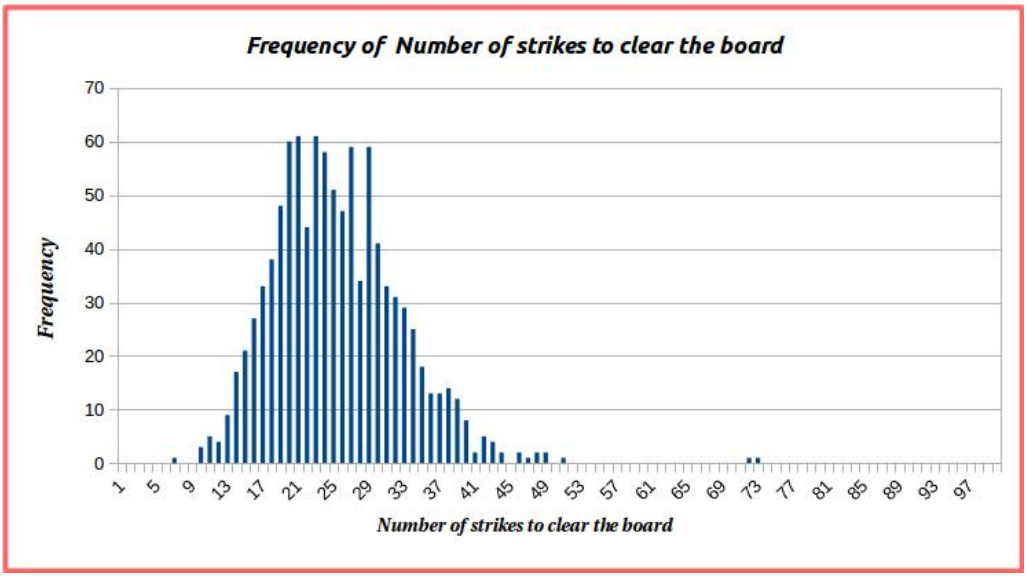

Figure 1: Frequency of number of strikes to clear the board in single player mode

Agents developed based on learning algorithms 1 and 3 did not yield good results. Most of the times, they are taking more than 40 moves to clear the board. But, when learning algorithm 2 (predicts the coin to be hit) is used to train the network and

making the agent use the deterministic algorithm to pocket that coin, has proved to be better than other approaches. Using this agent, we played 1000 games and we plotted 'number of games completed' on y-axis and 'number of strikes it took to clear the board' on x-axis. Figure 1, summarizes the result of the 1000 games the agent played.
Median = 25 strikes
Mean = 25.428 strikes
Standard deviation $\leq$ 4 strikes
Nearly 80% of games are completed in less than or equal to 30 strikes.

5 Conclusion

Though we think that agent developed based on learning algorithm 1, should be more successful than others, our experiments are not able to prove our point. In this report we have discussed about our ideas and intuitions in developing an agent to play caroms game in an optimal way. Our experiments show that, the agent which we developed based on pocketing the coin, predicted by neural network is able to clear the board in 25 moves on average, with standard deviation less than 4.

6 References

1) Q-learning with Neural Networks
http://outlace.com/Reinforcement-Learning-Part-3/

2)Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto, MIT Press, 1998, http://webdocs.cs.ualberta.ca/ sutton/book/ebook/

3)Demystifying Deep Reinforcement Learning https://www.nervanasys.com/demystifying-deep-reinforcement-learning/

4)PyBrain - Neural network library http://pybrain.org/docs/