Team Name - AAA
CS747 Project Report

Anchit Gupta, Ankit Rathore, Aashish Rathi

13D100032, 130050029 , 130050016

November 15, 2016

One Player Agent

The goal of the one player agent is to clear the board in as few moves as possible.

Below are the some of the approaches we tried out for the 1 player agent. The final agent we submitted gave the best performance of an average of 24 turns to clear the board across 1000 runs.

Reinforcement Learning Approach

For our problem at hand, both the state space (positions of coins over 2-D boards) and action space(position of striker, angle to hit, force) are continuous. Having only a continuous state space and discrete action space can usually be dealt via function approximation and standard Q-learning approaches. But the fact that even our action space is continuous narrows down the choice of algorithms available to us.

However, we did briefly explore the approach where action space can be made discrete. But, given that we have a 3 dimensional action and the fact that the game-reward is very sensitive to small variations, for any reasonably fine discrete-sampled action space, the number of actions would blow up far too much and make the output layer of our deep-Q network too large and difficult to learn given the hardware constraints we have.

Deep Deterministic Policy Gradients

This is an algorithm proposed by Google Deep Mind which claims to deal continuous action spaces problems very efficiently. This is a policy gradient algorithm which uses the actor-critic framework. Both the actor and the critic are implemented as neural networks. To tackle the convergence problems when using neural networks in RL the algorithm uses Experience Replay along with a few other small tweaks. We referred [1] and implemented the same for the 1 player carom agent. Initially we just encoded the coin coordinates directly as the state and ran the algorithm for around 200 episodes (run time of 4 hours). We could not observe any appreciable convergence and the algorithm often gave the same action for very different states.

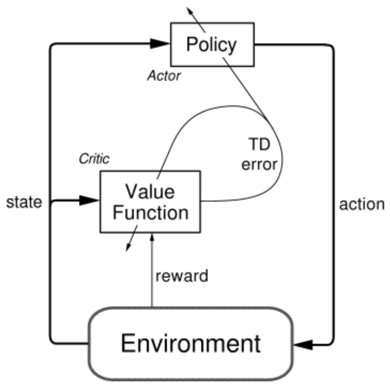

Figure 1: Actor Critic Framework

To try and mitigate this we tried making the state representation more rich, specifically given the angles, distances to the holes along with the coordinates for each coin. Sadly, even this did not result in any appreciable benefit.

As is evident in [1] even in simple games like Cart Pole thousands of episodes were needed for convergence. Given the very complex nature of our game we would definitely need even more episodes.

Learning Just the force

After the above method failed, we tried to reduce the no. of parameters learned. Now, the striker position and the angle to hit were now determined by our hand coded logic, and the force was made 100 levels discrete. We implemented a standard Q-learning approach [3] [4] to learn the best force to hit with given the current state. As is clearly evident from the graph below this, the method did result in some convergence as only the force need to be learned. However, when we checked most of the predictions, the network seemed to learn that hitting with highest force was the best way to go about on average. We were already doing the same in our hand coded approach which gave better performance than this method.

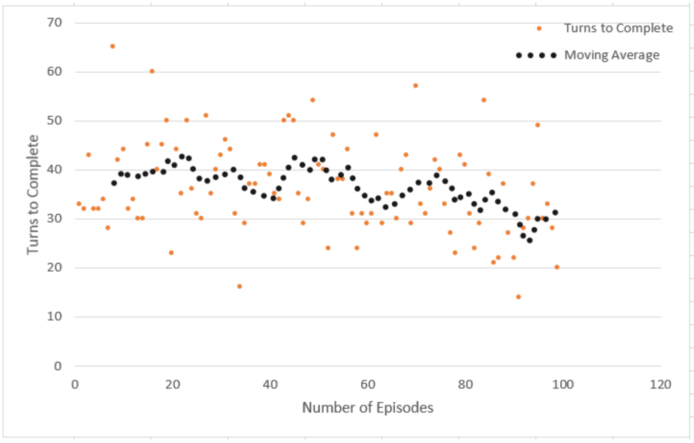

Figure 2: Force Learning Convergence

First Move Optimization

Our hand coded agent's performance seemed to depend a lot on the first move it made. If the queen was pocketed in the first move then it usually finished the game much quicker. As the first move is in some sense always in our control we tried to find out the best one averaged across various random seeds. We used a simplex like algorithm implemented in scipy to find the best first move.

Depending on where we started the optimization process using simplex we received 2 optimal actions. One of which outperformed the other when averaged across a larger number of seeds and hence was chosen for our final agent.

The table below summarizes the results, the first entry is the action we used in our old agent, and the other two are obtained using simplex.

Table 1: First Move Optimizations

Action	First move reward	Avg. Total turns
(1, 127.78, 0.5)	4.11	25.9
(1,123.82, 0.5)	4.84	25.7
(0.5,90,1)	5.48	25.2

Hard Coded Approach

Cluster Strike

After playing around with the simulator we discovered that hitting with a large force into a cluster of closely spaced coins often resulted in lots of coins being pocketed in one turn. This also closely follows what a human player would do during the beginning stages of the game when the coin density is high.

Hence, this approach mainly has three steps,

- Firstly, finding the coin which we want to hit using the formula

$$argmin_i \sum_j dist(coin(i), coin(j))$$

- Secondly, the best hole to aim for which is at at an allowable angle for a direct hit [-45,225] from the coin and is also closest to the coin among such holes.
- Then, for computing the position to place the striker we take into account the angle to the hole from the target coin as well as coins obstructing this direct path, if any.

The below graph is plotted for 1000 runs across different seeds.

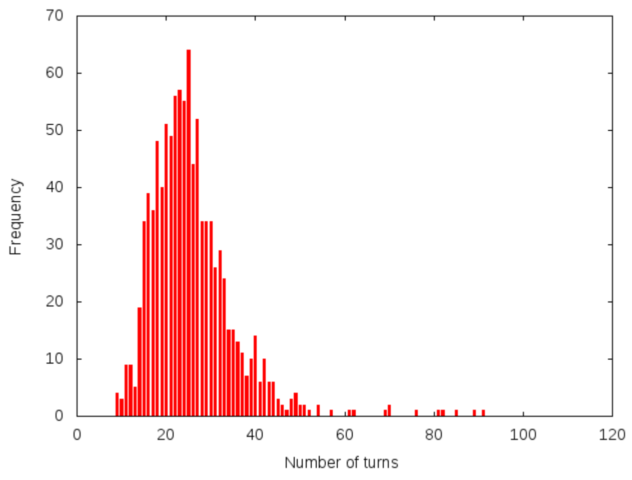

Figure 3: Agent v1.0 Average 25.701

Observations from graph

As is evident by this frequency distribution graph above we can improve the performance of our agent by reducing the median. Also the distribution is quite long tailed and those few runs with > 50 turns hurt the average score a lot. So we set out to both try and reduce the mode and also in some sense truncate the long tail of the distribution.

Precise Direct Hit

To tackle the tail we observed a few of the > 50 turn games and realized that our agent was playing very imprecisely just hitting it with large force and hoping that with luck the coins get pocketed while this works well in most cases for a few bad seeds this resulted in a sequence of bad moves. To combat this we implemented a precise direct hit logic which finds coins with an unobstructed path to the hole and hits them with a precisely controlled force. If no such coin is found we default to the earlier algorithm. This direct hit is only activated when the agent has already played 35 turns to try and combat the cases that result in the long tail. Using direct hit always instead of only for long runs results in a worse performance as the random noise added by the server is too much and often results in the turn being wasted by direct hit, and in such cases hitting hard is better.

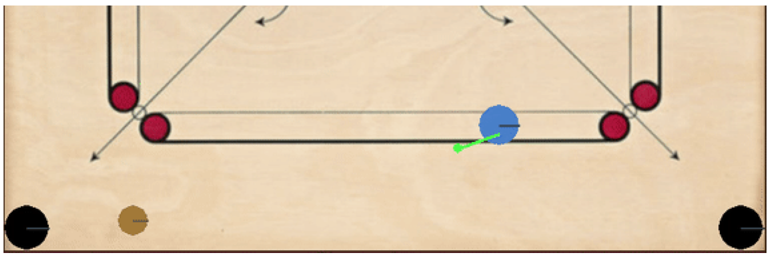

Figure 4: Direct Hit

Ghost Holes

Most of the logic of our agent especially where the angles to the holes, distances etc are calculated is done assuming the coin and the hole are singular points located at their respective centers. Though this clearly simplifies the task it is sub-optimal.

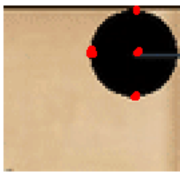

Figure 5: Location of Ghost holes

To combat this we introduced some ”Ghost holes” into out logic these new holes are points very close to the actual center of the holes so that hitting these would also result in the coins being pocketed. By doing this many new shots were now available to be taken and this successfully reduced the median of our distribution.

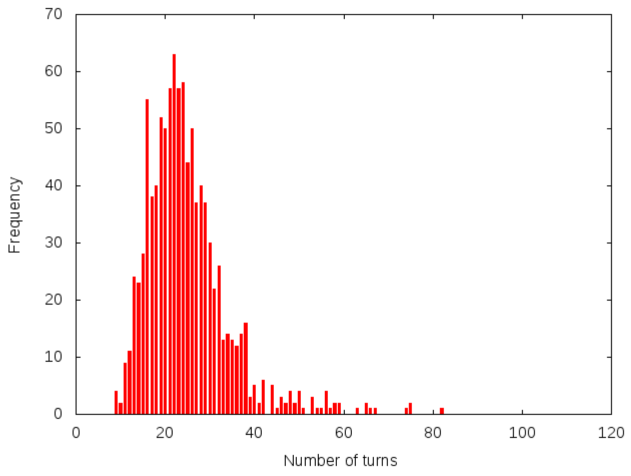

Figure 6: Agent v2.0 Average 24.81

It is evident from this graph that both the tail has been somewhat thinned and the median/mode also reduced.

Two Player Agent

Our goal here was to design an agent which could beat another agent in competitive play. In an ideal scenario our agent should take into account the moves the opponent will make in the future and hence play a move which minimizes that along with maximizing his own payoff. A method like Monte-Carlo Tree Search would we well suited for this. We would have needed a scoring function for states and some way to simulate a move. Given the time constraints of 0.5s set by the server and the slowness of the one step simulate function provided to us (0.4s per turn) this was not an option. Given our experience with RL methods on the 1 player agent and because of a paucity of time we went on to only implement a hand coded agent which tries to maximize its one step rewards.

v1.0 Agent

We started out naively with our 1 player agent making straightforward changes to the targeting and obstruction finding code so that only the required color coins are targeted and all coins considered for obstructions. The main complication here was due to one of the foul rules. If we accidentally pocket our opponents color then it is a foul and all coins pocketed in that turn are returned. This rule called for a very different approach to designing the agent. Focusing more on hitting precise shots rather that hitting it with a large force seemed to be the way to go .

v2.0 Agent

For this agent, we go through the basic flow-chart as follows :-

- We try to make a direct hit to one of our coins, if possible.
- If direct hit doesn't work, we make a cluster hit, considering only our own coins for cluster and all others for obstruction.

These tweaks when combined together gave us a performance improvement.

Table 2: Perfromance Analysis v1.0 vs v2.0

Matches	Win Percentage	Average Run-time
v2.0 against v1.0	81.73	45s
v1.0 against random	85.34	70s
v2.0 against random	90.18	62s

In most of the cases in which our agents lose to the random approach is due to the queen pocketing rule where we pocket all our coins before pocketing the queen and hence lose. Prioritizing the queen earlier in the game is one way to combat this in the future.

Conclusion

Though Deep Reinforcement Learning has been applied very successfully to play various games ranging from Atari to Go [5] given the inherent complexity of carrom being a continuous spaced game along with the limited hardware we have access to makes it difficult to achieve success with the same.
To make a successful Deep RL Bot in some sense the neural network needs to learn physics of the environment and have some ability to internally visualize various moves and chose the best amongst them. Interestingly in [2] for a very similar game of billiards the authors tried to make a much more complex network than ours learn the physics of the game and achieve simple tasks like pushing a single ball to a desired location. They ran experiments with upto three balls and were only able to achieve modest results of 50-60% accuracy for the task. This shows the difficulty current methods have with games like carrom.

References

[1] Patrick Emami. *Deep Deterministic Policy Gradients in TensorFlow.* URL: `http://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html`.

[2] Katerina Fragkiadaki et al. "Learning Visual Predictive Models of Physics for Playing Billiards". In: *CoRR* abs/1511.07404 (2015). URL: `http://arxiv.org/abs/1511.07404`.

[3] Arthur Juliani. *Simple Reinforcement Learning with Tensorflow.* URL: `https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0#.hqkl08w5m`.

[4] Andrej Karpathy. *Deep Reinforcement Learning: Pong from Pixels.* URL: `http://karpathy.github.io/2016/05/31/rl/`.

[5] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). URL: `http://arxiv.org/abs/1312.5602`.