# Intelligent Pool Playing Agent

**Atharv Nandapurkar**
160050004
atharvn@cse.iitb.ac.in

**Rajat Rathi**
160050015
rathi@cse.iitb.ac.in

**Tummidi Nikhil**
160050096
tummidi@cse.iitb.ac.in

**Gurparkash Singh**
160050112
gpssohi@cse.iitb.ac.in

## Abstract

We propose a Neural Network based Deep Q-Learning approach to build an agent that could autonomously learn to play the game of 8-Ball Pool. We also present a more constrained formulation of the game which makes our model more realistic as compared to the relatively relaxed constraints in the existing work. We describe an innovative Reward Function that provides an agent with more detailed information and context to learn from. We also conduct an analytical survey comparing the performances of various Reinforcement Learning algorithms on our modified environment.

## 1 Introduction

Our 8-Ball Pool setup has 4 types of balls on the table. The white ball is used by the player to hit using the cue. There are seven balls of each type: striped and solid. In our setup, the player chooses the type of balls that he will try to pot. The other player will try to pot all the balls of the opposite type. After a player has captured all the balls of the chosen type, he will try to sink the special "8-Ball". If a player sinks the 8-Ball before having sunk all the other balls of the assigned type, the player automatically looses. The player that legally sinks the 8-Ball wins. Sinking the white ball is a foul, which allows the opponent to place the white-ball anywhere on the board. However, for simplicity, the agent does not move the ball if the opponent commits a foul and instead starts playing from a default position. Sinking the ball of the assigned type results in a repeated turn. In order to make a move, our agent will have to decide the angle as well as the force with which to hit the white ball using the cue. This is what comprises the action in a given state.

Pool is a very complex game with several nuances that can make developing a generalized agent a challenging task. Therefore, before we can hope to build an omnipotent agent, it is wiser to relax some of the constraints and learn in a more simplified model of the game. However, the simplifications that we have specified are much closer to the original game as compared to the existing work. For the most part, our simplifications are just concerned with ignoring the fringe rules of the game, such as the first move is only valid if at least 4 balls hit the boundaries of the board. Ignoring such rules doesn't affect the overall working of the game but make its modelling more convenient. We believe that making a Pool playing agent has various complications that need to be dealt with which makes it a stimulating and interesting problem.

## 2 Existing Work

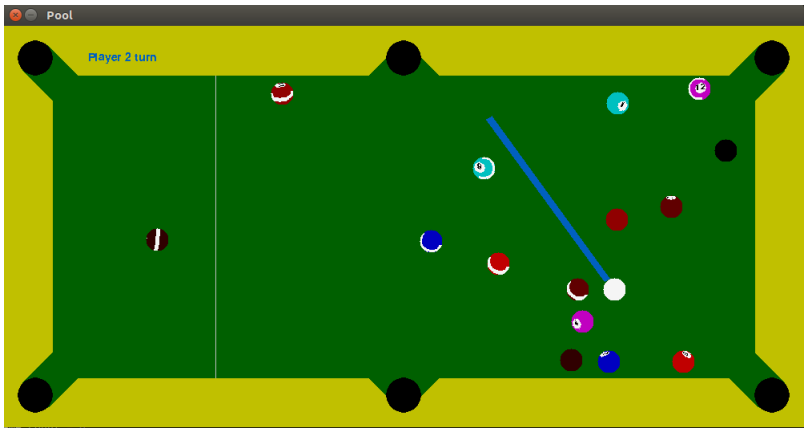
Most of the existing work on Pool playing Reinforcement Learning make assumptions that make the environment too simplistic. [6] uses Neural Networks to learn the Physics of the environment and use that information to give a score to each of the actions depending upon their predicted outcome in

the environment. [5] combines Neural Networks with Monte Carlo Search Trees to efficiently search through the state-action space to decide the optimal actions. We turn our focus on [1], the work we use as a benchmark and try to improve upon. The authors have developed a model that can learn to sink all the balls on the pool table sequentially. They have used a modified version of the original pool game simulator [3] [4] to train their model. In the paper, the agent has been formulated to treat all the balls equally and to sink them sequentially using the white ball. The reward function used by them is greedy in the sense that it makes the agent to try to sink the maximum number of balls in one turn. However, this might not be optimum, because, in the long run, it might be better to sink a fewer number of balls in the current turn and set up the positions of the other balls such that it takes fewer number of turns to sink all the balls on the table. Also, such a greedy strategy won't work in our environment because of the presence of opposition's balls. For example, a move that sinks only 1 ball but makes it more difficult for the opponent to score points might be a better strategy than sinking the highest number of balls but in turn, making it easier for the opponent as well. Furthermore, treating all the balls equally is far from reality as the real complication arises when the agent has to decide which balls to sink and in which order. The authors have done a comparative analysis of 3 algorithms, Actor-Critic Model, Table-Based Q-Learning, and Deep-Q Networks. We have tried to focus mainly on Deep-Q Networks because according to the authors, DQNs perform the best among the given algorithms. However, we also provide results of using Semi-Gradient SARSA for comparison.

## 3 Implementation Details

### 3.1 Simulator

We made [1] modifications over an existing Pool simulator to cater to our requirements. Firstly, we added an option to disable graphics rendering to facilitate faster training of the agent. We also made alterations to the base code to permit the agent to learn across multiple runs of the game, along with provision for interaction between the agent and the environment. Another feature that we implemented was allowance for any combination of bots and humans to play with each other. So, bots running with different algorithms can play against each other.




### 3.2 Reward Function Formulation

The reward function is implemented following the below rules

- If white ball is potted a negative reward of 1 is awarded
- If no ball is touched in a turn, then also a negative reward of 1 is awarded
- If an opponent's ball is potted, then a negative reward of 1 is awarded
- If 8-ball is potted earlier, a negative reward of 5 is awarded (agent loses the game)
- If 8-Ball is potted after all the other balls, a reward of 5 is awarded (agent wins the game)
- If the preferred ball type is potted, a positive reward of 1 is given
- Else no reward is given, i.e., -0.5

[1] https://github.com/Nikhil-t01/pool-agent

## 4 Methodologies

We have used three algorithms - Deep-Q Networks, Semi-Gradient SARSA, and a Greedy approach. Both the algorithms are Value-based and try to estimate the action values for the states. Coming up with the state-action modelling was fairly challenging, as discretizing the coordinates leads to a large number of states and actions. Also, designing features was tricky as it needs to take care of the fact that for each state, the best action might be different and should be compatible with a linear approximator. To be more specific, we need to account for the fact that just increasing the angle or force of an action should not linearly increase the q-value. Below, we explain in detail the methodologies employed by us corresponding to each method.

### 4.1 Deep Q-Networks

We use a Fully Connected Neural Network to map states to the Q-values of actions. The input layer is $d_1$-dimensional where $d_1$ is the number of state features. In our case, we have used 48 features. First 32 features corresponding to the locations (x,y) of all the 16 balls and the next 16 features are binary which indicates if the corresponding ball is in play or not. The output layer is $d_2$ dimensional where $d_2$ is the number of possible actions. Even though the action space is continuous, we have discretized it to 400 values. The angles take 20 possible values and the amount of force also takes 20 values. The cross product of these sets means that there are 400 possible actions. We experimented with several models and chose an architecture with 2 hidden layers with 64 and 128 neurons respectively. We used the PyTorch Library in Python to implement and train the Neural Network. The autograd feature of the library provides a convenient way of calculating gradients of the Network. During training, we use $\epsilon$-greedy (with an $\epsilon$=0.3 whereas while testing, we choose the action greedily with respect to the Q-Values. We used a learning rate of 0.05.

$$w \leftarrow w + \alpha[R + \hat{q}(S^{'}, A^{'}, w) - \hat{q}(S, A, w)]\nabla\hat{q}(S, A, w)$$

R is the reward at the current time-step and $\hat{q}(S^{'}, A^{'}, w)$ is the estimated Q-Value for a given state and action, as output by the neural network. $\nabla\hat{q}(S, A, w)$ is the gradient of the output node of the Network with respect to the weights, and is computed using the autograd feature. $w$ denotes the weights of the model.

### 4.2 Semi-Gradient SARSA

In this method, we used a linear function approximation for action-value function using a set of weights and features. The states are coordinates of each ball on the table. For a ball, on taking a particular action, if the move direction of weight ball is towards it, then the feature vector of that ball is set to 1 and other balls to 0. The weights of the model are learned by the semi-gradient SARSA update.

$$w \leftarrow w + \alpha[R + \hat{q}(S^{'}, A^{'}, w) - \hat{q}(S, A, w)]\hat{x}(S, A, w)$$

Where $w$ is the weight vector, $\hat{x}$ is the feature vector, $\alpha$ is learning rate, $\hat{q}()$ is the action-value function approximated as

$$\hat{q}(S, A, w) = w \cdot \hat{x}(S, A)$$

The feature vector $\hat{x}(S, A)$ is set as explained above. Since it is a linear approximation, the gradient of the action-value function is the feature vector itself

### 4.3 Greedy Approach

In this greedy approach, the bot always hits the closest preferred ball to the white ball. The major drawback of this approach is that there is no learning. It doesn't direct the ball towards any hole. The good side is that it at least hits the preferred type balls. Compared to others, results of this are not particularly good, as sometimes it might hit an opponent's balls or the 8-ball which is in the path of its closest preferred type ball.

## 5 Results

Along with implementing the above algorithms, for comparison sake we also included an agent that picks an angle and force uniformly at random. Firstly, we observed that as expected, each

of the above three approaches work better than the "random" agent. Semi-gradient SARSA with linear approximation and DQNs each require a significant amount of training before they begin to perform well, so in the beginning the "closest-greedy" agent outperforms the two. After hours of training, we began to notice that the two learning algorithms began to play better than the greedy approach, with the DQN agent yielding the best results, which aligns with our theoretical predictions that these algorithms would take future orientation as well into perspective when taking an action.

While we have presented a relative comparison between these methods, it is also important to note that our agent didn't reach a stage where the bot via any algorithm could keep up with average human performance (mean of the 4 humans who were involved in this project), who in expectation pots one ball for every two moves.

We had also experimented with some reward function formulations. For instance, earlier we had set zero reward for a move when the cue ball touches some balls but none are potted. This led to very slow learning, so we set a -0.5 reward to incentivize the agent to pot balls more quickly, which gave quicker results.

## 6 Conclusion

We have shown that it is possible to improve the performance of an agent by choosing a more informative reward function. Furthermore, we conclude that using Neural Networks gives a better performance as compared to Linear as well as Greedy approach. Furthermore, Neural Networks also provide a convenient way to incorporate continuous states.

## 7 Future Scope

Various constraints that we have relaxed in our implementation of pool can be improved further. Some of such relaxations include

- We have made our player choose between stripes and solid ball before the start of the game, whereas in a real game a player chooses to depend upon match scenario.
- The first shot is valid only if at least four balls touch the perimeter of the pool.
- It is necessary to declare the hole in which the player is going to pocket the 8-ball before attempting to sink it. This constraint has been relaxed in our implementation.

So far, all of the existing work has been in the domain of Value-estimation based algorithms. However, policy-iteration based algorithms, such as Reinforce and Roll-Out can be tried. [5] show that combining Monte Carlo Tree Search with Neural Networks can improve the performance. Therefore, we can try that paradigm for training agents as well. The success of Actor-Critic methods [7] in various Reinforcement Learning tasks is an indication of the fact that they can prove successful in the 8-Ball Pool environment as well. Using Actor-Critic approaches, we can also incorporate continuous actions which have the potential to further improve the performance.

## References


[1] Peiyu Liao et. al., Deep Cue Learning: A Reinforcement Learning Agent for Playing Pool

[2] Implementation of the above paper (`https://github.com/nkatz565/CS229-pool`)

[3] Pool simulator used in paper (`https://github.com/max-kov/pool`)

[4] Pymunk library (`http://www.pymunk.org`)

[5] Yu Chen & Yujun Li, Macro and Micro Reinforcement Learning for Playing Nine-ball Pool

[6] Katerina Fragkiadaki et. al., Learning Visual Predictive Models of Physics for Playing Billiards

[7] Ivo Grondman et. al., A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients