Macro and Micro Reinforcement Learning for Playing Nine-ball Pool

1st Yu Chen
Guanghua School of Management
Peking University
Beijing, China
yu.chen@pku.edu.cn

2nd Yujun Li
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China
liyujun145@gmail.com

***Abstract*—We present a method of training a reinforcement learning agent to play nine-ball pool. The training process uses a combination of reinforcement learning, deep neural networks and search trees. These technologies have achieved tremendous results in discrete strategy board games, and we extend their applications to pool games, which is a complicated continuous case. Pool types of games have a huge action space, to improve the efficiency of exploration, we use a macro and micro action framework to combine reinforcement learning and the search tree. The agent learns skills such as choosing pockets and control the post-collision position. Our method shows the potential to solve billiards planning problems through AI.**

***Index Terms*—reinforcement learning, nine-ball pool, Monte Carlo tree search**

I. Introduction

Various games provide rich simplified testbeds for reinforcement learning (RL) algorithms [1]–[3]. Moreover, RL can discover new and creative ways of playing games [4], [5]. The research on the two mutually beneficial fields has a long history [6], [7], and a lot of breakthroughs have recently emerged due to the development of deep neural networks.

Among the RL applications in games, there are three main categories, namely board games, video games, and physics based games. In this work, we apply reinforcement learning to the billiards game. Billiards game contains a variety of physical movements, including sliding, rolling and collision processes. At the same time, it also requires global planning like board games; for example, the player is required to control the cue ball position for the next shot.

A. Nine-ball

Nine-ball is the predominant professional game in the World Pool-Billiard Association. It is played on a billiard table of ten balls, with one cue ball and nine colored balls labeled from 1 to 9, respectively. The objective is to pocket the balls in ascending order, from 1 until 9, and the player that pockets the last ball wins the match. Therefore, players need to pocket target balls successively until the final ball.

B. The Challenge

As we mentioned above, cue sports require precise physical skills as well as global planning. Classic games, such as Go, Chess and Texas Hold'em, put more emphasis on planning. Billiards are similar; players need to decide what to do, for example, what is the best pocket to put the target ball in. To accomplish the goal, players also need to make decisions such as the speed, angle and spinning methods (Changing the collision point to rotate the ball).

Games such as Go and chess have detailed human replay data, while for billiards the data is insufficient. It is unfeasible to perform supervised learning by imitating human players. Besides, accuracy is especially crucial in nine-ball. For example, the aiming angle takes a value between 0 and 360 degrees, but 0.3-degree error may lead to a failure shot.

C. Related Work

There are several AI eight-ball players [8]–[11]. They use a Monte Carlo search for shots generation and use domain knowledge to improve performance. Nine-ball is professional pool game and is generally accepted to be more difficult than eight-ball, where the latter significantly reduces the requirements of planning.

We follow the reinforcement learning framework in AlphaGo [4]. Regarding the characteristics of billiards, a macro and micro action structure is used to speed up training.

II. The Learning Environment

We trained the artificial nine-ball player in a simulated environment based on the work of [12], which uses explicit physical models for the balls, cue, table and their interactions. The methodology was also used as the physics model in the Computational Pool Tournament of the 10^{th} Computer Olympiad.

State We use a three-dimensional coordinate (x, y, z) to represent a ball's position in the environment. (x, y) are the position on the table, and z is the height from the ground, which indicates whether the ball has been pocketed. State s consists of coordinates of the ten balls. s can be indexed by a sequence of discrete time points $t = 0, 1, 2, \cdots$, where s_0 represents the initial state before the break shot, and s_t represents the state before t-th shot.

Action Space We follow previous studies in robotic billiard, for example, [8], [10], [12] and use five continuous

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

parameters for an action. The first three parameters represent the movement of the cue stick, namely,

ψ, cue stick horizontal angular, i.e. the aiming direction.
ϕ, cue stick vertical angular, i.e. the elevation of the cue.
v, initial speed of the stick.

Hitting the cue ball at an off-center point will cause the ball to spin, which is an important skill. The related parameters include,

w, the horizontal offset from the center.
h, the vertical offset from the center.

Rewards According to the nine-ball rule, winning occurs when a player pockets the final ball without committing a foul. Hence, $r_t = 1$ if the final ball is pocketed, $r_t = 0$ if the other balls are pocketed. If the agent fails to pocket the target ball, $r_t = -1$. In this work, the agent is trained to maximize the winning probability.

III. Macro and Micro Action Reinforcement Learning

As mentioned previously, nine-ball requires both precise shooting and global planning. In this section, we demonstrate a novel method to train a nine-ball player[1].

A. Macro and Micro Actions

With the help of cushions, we have many ways to pocket the target ball (see Fig. 5 for example). If we link the path of which the cue ball moves to the target ball, and the path of which the target ball to its pocket, we will get a polyline called aiming path. Some example aiming paths are presented in the second and third row of Fig. 2.

According to the characteristics of billiards, scholars use a two-step action framework for billiards AI in the literature [9], [10]. We divide the action parameters into macro actions A_t and micro actions a_t respectively. Our contribution is to introduce reinforcement learning into this framework to improve global planning.

The macro action is the choice of aiming path, i.e., the shooting angle ψ. For each s_t, let $\Omega(s_t)$ be the set of candidates ψ. $\Omega(s_t)$ includes different shooting options, such as different pockets, whether to use cushions.

To adjust the post-collision position of the cue ball, we need micro actions to "fine tune" how we shoot. The micro action is $(\Delta\psi, v, \theta, w, h)$, where $\Delta\psi$ is the adjustment added to ψ in the macro action. The micro action determines how to shoot, for example how far the cue ball can go and whether the cue ball will follow or draw.

B. Actor and Critic Functions

A Q-function $Q(s_t, A_t)$ is used for macro value estimation, where A_t is the macro action from $\Omega(s_t)$. We use a policy function $\pi(s_t, A_t)$ to produce a_t. $\pi(s_t, A_t)$ is as function of A_t, because the micro action needs to adjust according to macro action, for example for those A_t with long aiming path, a larger speed is needed consequently.

[1]See a video demo at https://youtu.be/icHWGVhCTKs

TABLE I: Input features for $Q(s_t, A_t)$ and $\pi(s_t, A_t)$.

Feature	#	description
Position	6	Cue ball, target ball, next three target balls and all balls
Aiming Path	2	Aiming path for A_t; paths from final ball to each pocket
Table	2	Contour of cushions and pockets

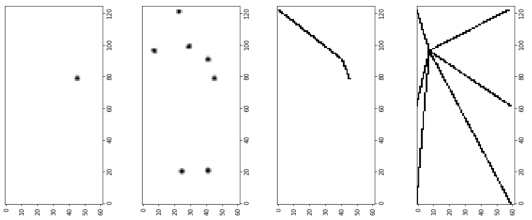

Fig. 1: Feature plane samples in nine-ball for the rendered position maps and aiming paths.

The environment provides the coordinates as well as the rendered image for state representations. We used 10 images, each representing different information of the game, for example, the cue ball position and the target ball position. All images are resized to 62×125 for the convenience of calculation. Feature details are listed in Table I and see Fig. 1 for a comprehensive understanding. We choose the image representation because it can better capture relative position information.

Q and π share a common CNN, which is designed as follows,

two layers of 64 kernels with size 7×7 and stride 3
two layers of 128 kernels with size 5×5 and stride 1
two layers of 256 kernels with size 3×3 and stride 1
a dense layer of 512 kernels

C. Use Tree Search to Train and Inference

Similar to Chess and Go games, we conduct planning for nine-ball by a search tree which inherits the framework of [4]. There are two types of nodes in the search tree, namely common nodes and sub-nodes. A common node represents a real state of the game, while a sub-node stands for an imaginary aiming plan, which is a pair of state and macro action. Each common node s_t contains $|\Omega(s_t)|$ sub-nodes (s_t, A_t), corresponding to different aiming paths. For every sub-node, the visit count $N(s_t, A_t)$ is recorded, and a micro action is proposed by the actor function $\pi(s_t, A_t)$ with an added Gaussian disturbance, which is referred to as $\omega(s_t, A_t) = \pi(s_t, A_t) + \epsilon$. Each sub-node connects to its children states s_{t+1} by a sequence of exploring micro actions based on $\omega(s_t, A_t)$. The value of node s_t is the maximum of its sub-node children, i.e, $\max_A Q(s_t, A)$, and the value of a sub-node (s_t, A) is the maximum of value of next state s_{t+1}. See Fig. 2 for a graphical presentation for the procedure.

Tree select policies During the tree search procedure, the macro action is selected by an upper confidence bounds (UCB) method, and the micro action is by a renewal exploring

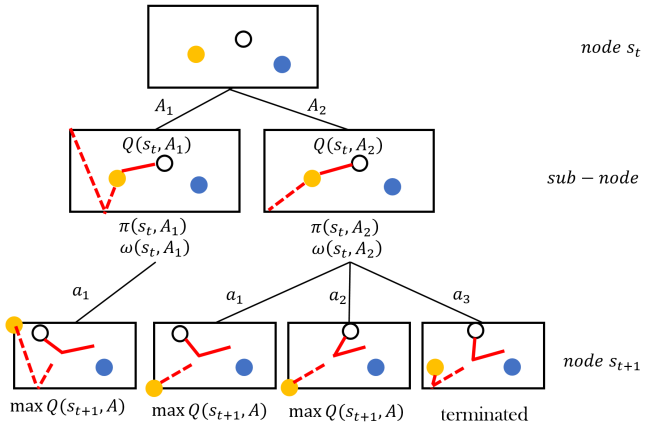

Fig. 2: The tree search procedure for nine-ball. For each node s_t, a macro action is selected first and then followed by a micro action based on $\pi(s, A)$ and the exploratory set $\omega(s, A)$. The value of each state is evaluated by $Q(s, A)$.

method, which is discussed later. For a given state s_t, select A_t that maximize $\alpha Q(s_t, A_t) + (1-\alpha)U(s_t, A_t)$ where the first part is used to drive the algorithm to exploit by estimated values, and the second is for macro actions exploration. $U(s_t, A_t)$ is defined as $\sqrt{\sum_A N(s_t, A)}/N(s_t, A_t)$, representing compensation for sub-nodes that have not been explored, i.e., drive the algorithm to explore the less visited nodes.

Renewal Exploring Method We develop a novel mechanism to determine the number of exploring trials for tree search with continuous policy, which is described in Algorithm 1.

Algorithm 1 Renewal Exploration Algorithm

Input: s_t, $\pi(s, A)$ and $Q(s, A)$.
Initialize $Q = -1, l = l = 0$.
while True **do**
  $l \leftarrow l + 1$
  Select A_t according to the UCB like method, evaluate $\pi(s_t, A_t)$, and set $a_t = \pi(s_t, A_t) + \delta$, where δ is a Gaussian disturbance
  Get s^0_{t+1} from the environment and set $Q = \arg\max_A Q(s^0_{t+1}, A)$
  if $Q - Q > 2 \times \text{SD}(\{Q_i \mid l < i \le l\})/\sqrt{l - l}$ **then**
    $Q \leftarrow Q, l \leftarrow l$
  else if $l - l >$ MAXSTEPS **then**
    break

The algorithm is inspired by the renewal process, and it includes two mechanisms, namely stopping and refreshing. If the algorithm cannot explore better results for a long time, the process will stop. Otherwise, the whole process is refreshed when reaching a significantly better result according to the t-test of Q values.

Self-playing policies The search tree stops when the renewal exploration algorithm stops, and then the agent selects an action and sends it to the simulator. The macro action A_t is chosen by maximizing $Q(s_t, A_t)$, and a_t is selected by maximizing $\max_{A_{t+1} 2 \Omega(s_{t+1})} Q(s_{t+1}, A_{t+1})$, where s_{t+1} is the simulated result of acting (A_t, a_t), i.e., the micro action is chosen among the sub-node (s_t, A_t)'s children, by maximizing the $t+1$ state value.

Parameters update From $t = 0$ (the break shot) to T (misses, commits a foul, or pockets the final ball), the agent collects r_t from the simulator, and updates $Q(s_t, A_t)$ from the environmental feedback. The total loss consists of two parts, $l_{total} = (l_A + l_a)/2$, where the loss for macro action l_A is $(Q(s_t, A_t) - (r_t + \gamma \max_{a_t, A_{t+1}} Q(s_{t+1}, A_{t+1}))^2$, and the micro action loss l_a is $\|\pi(s_t, A_t) - a_t\|_2^2$, i.e., micro actions are updated by supervised learning of the best tree search trials. We use the Adam algorithm [13] to update all parameters.

Training details The hyper parameters in the search process are set as follows, γ and α is set to be 0.95, the max steps in Algorithm 1 is set to be 200, the learning rate is 1×10^{-4}, the batch size is 32 and parameters of Adam algorithm is $(0.9, 0.999, 1 \times 10^{-8})$.

The training process is distributed on 16 CPUs and 2 GPUs. The billiards simulator on CPUs continuously send its state to the search tree, and the search tree will propose an action based on current network parameters and previous simulations. One GPU is used to update the parameters based on the new simulated data periodically, and the other one synchronizes with it and provides Q and π for the search tree. The whole system simulates 80,000 nine-ball games each day.

IV. Empirical Analysis and Performance

Networks training results We record the curve of the loss function over time. A simulated data queue is maintained, the network takes the latest 200,000 (s_t, A_t, a_t, R_t) tuples from the queue every hour to update the parameters, perform parameter updates, and records the corresponding losses. The losses curve are reported in Fig. 3. The $Q(s, A)$ contributes the most of the total loss. The training curve begins to converge after 30 hours.

AI Player's behavior Next, we analyze the behavior of the artificial player. The player executes 6,000 shots with the latest model. We present several statistics in Fig. 4 and an example replay in Fig. 5.

Comparison with the MCTS Since there are no publicly available nine-ball AI players, we choose the pure MCTS player as a comparison. The MCTS player does not have a

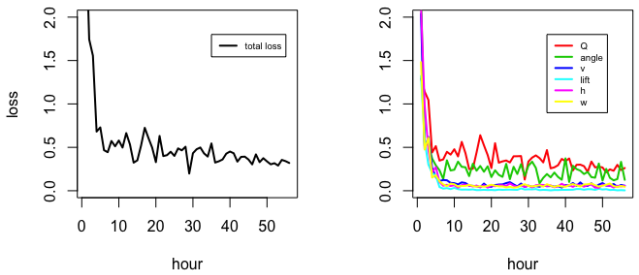

Fig. 3: The subgraph on the left shows the curve of l_{total} and the right one shows its components.