AutoBilliard: A Self-playing Billiard-like Game

Yiming Zhou, yxz180048@utdallas.edu

November 25, 2020

1 Introduction

This project implements a billiard-like game where only two kinds of balls competing each other. I call these two kinds are attacker and defenser. In this game setting, alongside the balls, there is also a gain field on the playground that the attackers try to reach to win the game. And the defensers should collide the attackers to defense the gain field. Denfensers fail if one of the attackers reach the gain field, vice versa.

1.1 Description

Suppose there are N attackers and M defensers in the playground. Each of them has properties position $p \in \mathbb{R}^2$, velocity $v \in \mathbb{R}^2$ and acceleration $a \in \mathbb{R}^2$. Though they are in the 3D world, the degree of freedom is two since they are only allowed to move on a flat plane. With every T unit time elapsed, an external force f is applied to each ball existing for T_1 unit and it disappears for the remaining T_2 unit, where $T = T_1 + T_2$. Alongside the external force, the friction force is accounted constantly. Denote the playground as $G = \{p | \forall p \in \mathbb{R}^2 \ and \ p \in Playground\}$, also the gain field as $G_{gain} = \{p | \forall p \in \mathbb{R}^2 \ and \ p \in G \ and \ p \in GainField\}$. For each property of the ball, use the superscript denotes the timestamp and the subscript denotes the index of the ball. For example, p_1^1 denotes the position of the ball indexed 1 at time 1.

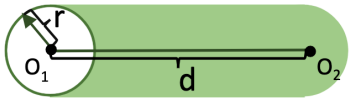
2 Method

Two major components of this game are 3D game world implemented with OpenGL and the corresponding moving strategies controlling the external force.

2.1 3D Game World

The playground in this game is essentially a rectangular shaped box bounded with walls and the gain field is a rectangular patch within the box. All the balls, attackers or defensers, are represented as spheres with the same mass and size.

Figure 1: Swept Area



2.1.1 Collision Detection

For the simplicity of the shape of the balls - sphere, the collision detection process can be simplified to detecting objects intersecting with the swept volume. Considering that the balls move on the plane horizontal to Z-axis, the detection of the swept volume is equal to the detection of the swept area shown as the bright green area in figure 1 between two sequential discrete timestamps t_k and t_{k+1}. The ball's center is positioned at O_1 at t_k and at O_2 at t_{k+1}, traveling distance of d. To detect whether any ball indexed i collides with the current ball, one needs to decides whether its center position p_i^k's distance towards the swept area is smaller than $2r$. Denote the vector pointing from O_1 to O_2 as v_{O_1O2} and the line function cross O_1 and O_2 as $F_{O_1O_2}$. The first condition should be satisfied is according to equation 1, where *distance* function computes the distance between the line and the point. The second condition should be satisfied is either one of equation 2 or equation 3, and *dot* is dot product operation.

$$distance(F_{O_1O_2}, p_i^k) < 2r \tag{1}$$

$$dot((p_i^k - O_1)v_{O_1O2}, (p_i^k - O_2)v_{O_1O2}) \leq 0 \tag{2}$$

$$|p_i^k - O_2| < 2r \tag{3}$$

2.1.2 Kinetic Dynamics

Ideally, the movement of each ball proceeds continuously, thus the chance of one ball colliding with more than one ball is pretty low. In such case, considering one colliding with the others not simultaneously but sequentially would not introduce much error to the result. Though it needs to be processed in discrete time in implementation where it is common that one ball has to collide with more than one ball simultaneously which makes the speed update very difficult, the current ball indexed i collides with n balls with indexes j_1, j_2, ..., j_n from near to far within the swept area and the update processes sequentially meaning velocity of ball i is updated after colliding with j_1 and the updated velocity is used for the next update with j_2 and so on. The detailed method is shown in algorithm 1.

For physical details, all collisions are considered to be elastic, meaning no kinetic energy loss. Since the momentum and the kinetic energy are conserved during the collision process, the following equation 4 and equation 5 are sufficed, where

v_i^- refers to the velocity of the ball indexed by i before the collision and v_i^+ refers its velocity after the collision.

Algorithm 1: Collision Update Algorithm

```
for i=1,...,N+M do
    v_i ← ball_i.get_speed();
    p_i ← ball_i.get_position();
    m_i ← ball_i.get_mass();
    for j=1,...,N+M do
        if i≠j and collides(i, j) then
            v_j ← ball_j.get_speed();
            p_j ← ball_j.get_position();
            m_j ← ball_j.get_mass();
            n ← normalize(p_i − p_j);
            v_1 ← dot(v_i, n);
            v_2 ← dot(v_j, n);
            v'_1 ← ((m_i − m_j)v_1 + 2m_j v_2)/(m_i + m_j);
            v'_2 ← ((m_j − m_i)v_2 + 2m_i v_1)/(m_i + m_j);
            v_i ← v_i + n(v'_1 − v_1);
            ball_j.update_speed(v_j + n(v'_2 − v_2));
        end
    end
end
```

$$\sum_i^{N+M} m_i v_i^- = \sum_i^{N+M} m_i v_i^+ \tag{4}$$

$$\sum_i^{N+M} \frac{1}{2} m_i (v_i^-)^2 = \sum_i^{N+M} \frac{1}{2} m_i (v_i^+)^2 \tag{5}$$

2.2 Strategy

The external force applied to each ball decides the way it moves. A chronological sequence of forces to each ball determines how it achieves the objective and such sequence of forces is the strategy. For the attacker, its objective is reaching the gain field which is trivial to make up a heuristic strategy that adds forces towards the gain field. Though it might collide with the other balls which makes it not the most efficient strategy, it is still good enough to reach the goal within short sequence of forces. On the other side, the defenser, it is much more difficult to figure out a heuristic method to "fight" back the attackers since it has to predict the path of each attacker first then choose the one that it has to fight back. The potential collisions occur during running makes it much harder. A modern method, deep reinforcement learning, is used to learn the strategy for the defenser by iteratively playing the game and learn by itself.

2.2.1 Heuristic Method

In the perspective of the attacker indexed i, at the time t_k the external force needed to be applied, it first calculates the unit direction vector from its center of mass p_i^k towards the center of mass of the gain field p_{cm_gain}. Then multiplying the direction with a constant scale gives the next external force f_i^k.

2.2.2 Deep Reinforcement Learning Method

The game is modeled as a multi-agent partially observable Markov Decision Process defined by tuple $(\mathcal{I}, \{\mathcal{A}_i\}, \mathcal{S}, O, R, \pi)$, where $\mathcal{I}$ is a finite set of agents, $\mathcal{A}_i$ is the set of actions for agent i, $\mathcal{S}$ is the set of environment states, O is the observation function, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function and $\pi : \mathcal{S} \to \mathcal{A}$ is the policy function. Given a trajectory $h_{1:T}$ of states, actions and rewards of length T defined over $\mathcal{S} \times \mathcal{A} \times \mathbb{R}$ space, $h_{1:T} = s_1, a_1, r_1, ..., s_T, a_T, r_T$, the total discounted reward at time t is defined as $r_t^\gamma = \sum_{k=t}^{\infty} \gamma^{k-t} R(s_k, a_k)$ where $0 < \gamma < 1$. The value function $V^\pi(s) = \mathbb{E}[r_1^\gamma | S_1 = s; \pi]$ is defined as the expected reward given initial state s and policy π. The Q-value function $Q^\pi(s, a) = \mathbb{E}[r_1^\gamma | S_1 = s, A_1 = a; \pi]$ is defined as the expected reward given initial state s and action a on such state with policy π. Also denote transitioning probability from state s to s' after time t as $p(s \to s', t, \pi)$. The discounted state distribution is defined as $\rho^\pi(s') = \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} p_1(s) p(s \to s', t, \pi) \mathrm{d}s$ where $p_1(s)$ is the initial state density of s. The performance objective with respect to polity π parameterized with θ is defined as equation 6.

$$\begin{aligned} J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \pi_\theta(s, a) R(s, a) \mathrm{d}a\mathrm{d}s \\ &= \mathbb{E}_{s \sim \rho^\pi, a \sim \pi_\theta}[R(s, a)] \end{aligned} \tag{6}$$

The policy gradient theorem[5] in equation 7 gives the gradient direction of objective function J with respect to parameters θ of policy π_θ enabling the updating process of policy parameters.

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s, a) \mathrm{d}a\mathrm{d}s \\ &= \mathbb{E}_{s \sim \rho^\pi, a \sim \pi_\theta}[\nabla_\theta log \pi_\theta(a|s) Q^\pi(s, a)] \end{aligned} \tag{7}$$

From equation 7 above, sampling $Q^\pi(s, a)$ from unknown true policy π is unreachable, instead, the critic-actor algorithm use Q^w function which is differentiable function with respect to parameters w to approximates the Q^π function. The w is chosen to minimize the error in equation 8 between true Q-value function and the estimated Q-value function to eliminate the bias. And the equation 9 gives the gradient substituting Q^w.

$$\epsilon^2(w) = \mathbb{E}_{s \sim \rho^\pi, a \sim \pi^\theta}[\nabla_\theta log \pi_\theta(a|s) Q^w(s, a)] \tag{8}$$

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)Q^w(s,a)] \tag{9}$$

It is useful to use off-policy[1] trajectories to estimate the policy gradient in which a different behavior policy β is used instead. Equation 10 gives objective performance under this setting.

$$J_\beta(\pi_\theta) = \int_{\mathcal{S}}\int_{\mathcal{A}} \rho^\beta(s)\pi_\theta(a|s)Q^\pi(s,a)\mathrm{d}a\mathrm{d}s \tag{10}$$

Applying the differentiation to the above objective function with approximation gives the off-policy policy gradient.

$$\begin{aligned}\nabla_\beta J_\beta(\pi_\theta) &\approx \int_{\mathcal{S}}\int_{\mathcal{A}} \rho^\beta(s)\nabla_\theta(a|s)Q^\pi(s,a)\mathrm{d}a\mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\beta, a\sim\beta}[\frac{\pi_\theta(a|s)}{\beta_\theta(a|s}\nabla_\theta log\pi_\theta(a|s)Q^\pi(s,a)]\end{aligned} \tag{11}$$

In the case of deterministic policy[4], a simple way to update the policy parameters is move the parameters in the direction of the gradient of Q-value, and the update rule is shown in equation 12, where μ is the deterministic policy.

$$\theta^{k+1} = \theta^k + \alpha_\theta \mathbb{E}_{s\sim\rho^{\mu^k}}[\nabla_\theta Q^{\mu^k}(s, \mu_\theta(s))] \tag{12}$$

The Sarsa update is used to estimate the critic function, i.e. Q-value function in the following equation 13, 14.

$$\delta_t = r_t + \gamma Q^w(s_{t+1}, a_{t+1}) - Q^w(s_t, a_t) \tag{13}$$

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q^w(s_t, a_t) \tag{14}$$

According to Recurrent DPG[2], it extends the observable state s to the history of sequence of observed states and actions, $h_t = (o_1, a_1, ..., a_{t-1}, o_t)$. Equation shows the new update rule for policy, where τ is entire trajectory $(s_1, o_1, a_1, s2, ...)$.

$$\theta^{k+1} = \theta^k + \alpha_\theta \mathbb{E}_\tau[\sum_t \gamma^{t-1} \frac{\partial Q^\mu(h_t, \mu^\theta(h_t))}{\partial\theta}] \tag{15}$$

Til now, we have discussed to formulate and update actor and critic function for single agent system. To extend the single agent system to multi-agent system, the actor function receives single agent's local observation to predict the next action while the Q value function receives the joint actions and joint observations to computes the action-state value. The actor and critic networks are implemented with recurrent networks. The actor networks share parameters among the agents.

Additionally, for the sake of stable training, target networks[3] are used to update networks gradually. To explore the state space, a noise is added to the

action every time the agent proceeds. The experience replay is deployed to fully train the networks.

Algorithm 2: Multi Agent Recurrent Deterministic Policy Gradient

Initialize critic network $Q^w(a_t, h_t)$ and actor network $\mu^\theta(h_t)$ with parameters w and θ.
Initialize target network $Q^{w'}$ and $mu^{\theta'}$ with weights $w' \leftarrow w$, $\theta' \leftarrow \theta$.
Initialize replay buffer R.
for *eposide=1,...,E* **do**
  Initialize empty history h_0
  for *t=1,...,T* **do**
    Receive environment state s_t
    $h_t \leftarrow concat(h_{t-1}, s_t)$
    Initialize empty action a_t
    for *i=1,...,N* **do**
      Choose action $a_t^i \leftarrow \mu^\theta(O_i(h_t)) + \epsilon$ (O_i, observation function for agent i; ϵ, exploration noise)
      $a_t \leftarrow concat(a_t, a_t^i)$
    end
    Apply a_t to environment
    $r_t \leftarrow R(s_t, a_t)$
  end
  Store sequence $(s_1, a_1, r_1, ..., s_T, a_T, r_T)$ in R
  Sample M trajectories $(s_1^i, a_1^i, r_1^i, ..., s_T^i, a_T^i, r_T^i)_{i=1,...,M}$ from R
  Construct M tracks of history $(s_1^i, a_1^i, ..., s_T^i, a_T^i)_{i=1,...,M}$
  Construct target value for each timestamp of each history:
  $y_t^i = r_t^i + \gamma Q^{w'}(h_{t+1}^i, \mu^{\theta'}(h_{t+1}^i))$
  Update critic network:
  $w := w - \alpha_{critic} \frac{1}{MT} \sum_i \sum_t (y_t^i - Q^w(h_t^i, a_t^i)) \frac{\partial Q^w(h_t^i, a_t^i)}{\partial w}$
  Update actor network:
  $\theta := \theta + \alpha_{actor} \frac{1}{MT} \sum_i \sum_t \frac{\partial Q^w(h_t^i, \mu^\theta(h_t^i))}{\partial a_t^i} \frac{\partial \mu^\theta(h_t^i)}{\partial \theta}$
  Update target critic: $w' \leftarrow \tau w + (1-\tau) w'$
  Update target actor: $\theta' \leftarrow \tau \theta + (1-\tau) \theta'$
end

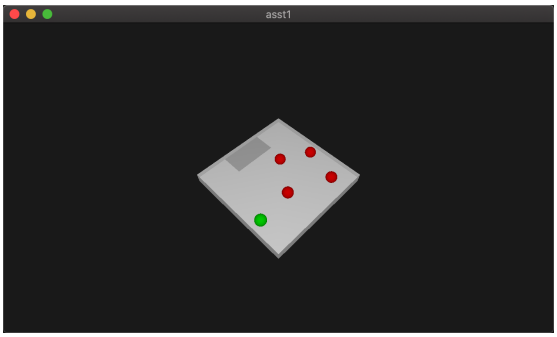
3 Result

The following works are done, and they can be summarized in section 3.1 and section 3.2. Code can be found in github[1].

- Implement the 3D models: planes and spheres
- Implement the collision detection algorithm for ball-ball collision and ball-wall collision

[1] `https://github.com/DonaldRR/CS6323`

Figure 2: Game World



- Implement the collision dynamics - speed changes after collision
- Implement the attacker's heuristic strategy
- Implement deep reinforcement learning framework for strategy learning for the defenser

3.1 3D Game World

The following figure 2 shows the implemented 3D game world. It is composed of a white box-shaped playground, a grey patch gain field and attackers and defensers which are represented by red and green balls respectively. The demo video[2] visualizes the collision process.

3.2 Strategy Performance

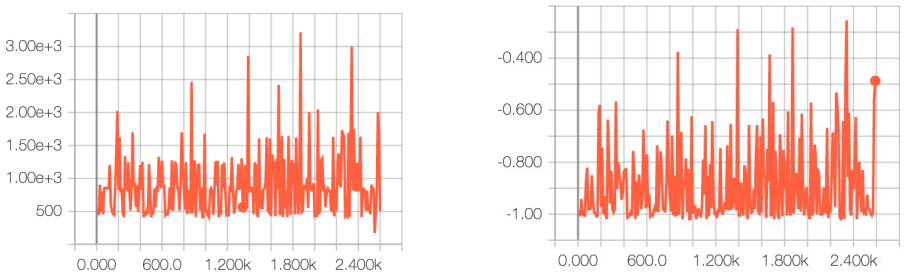
The game stops playing once one of the attackers reach the gain field. The shorter the game lasting time, the better the attacker's performance and the weaker the defensers' performance, vice versa. The figure 3 shows the game lasting time and the discounted reward in evaluation during during the training respectively.

4 Analysis

The novel contribution is the multi agent reinforcement learning task implemented in this project. Goals stated from the above section are meet except a steadily improved defensing strategy. The complexities of this include:

- Large state space and action space due those space are continuous
- Limited communication among defensers since a good cooperation is needed for all the defensers to achieve the shared goal

[2] `https://github.com/DonaldRR/CS6323/blob/main/media/video_demo.mov`



(a) Playing time

(b) Discounted reward

Figure 3: Strategy Performance

- Complex physical model where frequent collisions can make the environment state pretty hard to predict in one sense that small variation of the action could largely vary the resulting environmen stat

The stated complexities makes it hard to optimization the networks.

References

[1] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *arXiv preprint arXiv:1205.4839*, 2012.

[2] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

[3] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[4] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.

[5] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12:1057–1063, 1999.