Learning Billiards Through Q-Learning

Doncey Albin
Autonomous Robots and Perception Group (ARPG)
University of Colorado - Boulder
Boulder, CO, USA
doncey.albin@colorado.edu

Anna Zavei-Boroda
Autonomous Robots and Perception Group (ARPG)
University of Colorado - Boulder
Boulder, CO, USA
anza4273@colorado.edu

***Abstract*—This paper discusses the development, implementation, and outcomes of a tailored Q-learning and deep Q-learning approach applied to a custom-made billiards game. We start by addressing the fundamental principles and proceed with the design and construction of the game environment, ensuring its compatibility with reinforcement learning methods. We then outline the development and execution of both Q-learning and deep Q-learning, emphasizing the primary distinctions and challenges unique to each method. Subsequently, we share the findings of our experiments, providing insights into the efficiency and possible applications of these algorithms within the realm of billiards and comparable game environments. Ultimately, this paper seeks to enhance the comprehension and progress of reinforcement learning techniques in the sphere of games and simulations. All code is available on GitHub: https://github.com/donceykong/billiards-rl.**

***Index Terms*—Q Learning, Deep Q-Learning, OpenAI Gym**

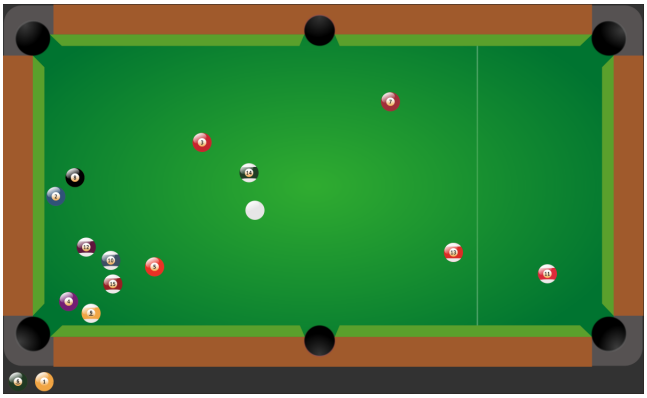

Fig. 1. Billiards Python game environment during game play.

I. Introduction

In recent years, reinforcement learning has emerged as a promising area of research, with numerous applications across various domains, including games and simulations [1]. Developing custom game environments and crafting original reinforcement learning implementations have become increasingly popular approaches in the field, as they provide students and researchers with opportunities to gain a deeper understanding of the underlying concepts and techniques [2].

This paper explores the benefits of such an approach, focusing on the design, implementation, and evaluation of a bespoke Q-learning algorithm applied to a custom-built billiards game (Fig. 1), as well as the inception and partial execution of deep Q-learning in the same environment. By referencing seminal works in the field, such as the DQN algorithm [3] and classic Q-learning by Watkins and Dayan [4], we demonstrate the value of custom game environments and original reinforcement learning implementations in fostering a comprehensive understanding of these techniques. Furthermore, we examine the challenges and insights gained from the development and application of these algorithms in the context of billiards and other similar game settings [5].

The paper is organized as follows: Section 1 introduces the foundational principles and design of the custom billiards environment, ensuring its compatibility with reinforcement learning methods. Section 2 explains the development of both Q-learning and deep Q-learning, emphasizing their key distinctions and challenges. Section 3 presents the implementation of our tailored Q-learning algorithm and its Q-table. Section 4 describes the deep Q-learning implementation for our custom game environment. Section 5 analyzes the experimental results, providing insights into the algorithms' effectiveness and potential applications in billiards and similar game environments. Section 6 concludes the paper by summarizing the contributions and exploring the broader implications of our findings for reinforcement learning techniques in games and simulations. Lastly, Section 7 suggests a promising direction for future research in this area.

II. Game Environment Design and Implementation

A. Environment Design

In our custom billiards game environment, we have carefully designed the state space, action space, and reward function to accurately represent the mechanics and objectives of the game while facilitating efficient learning for our Q-learning agent.

1) State Space: The state space of the billiards game environment encompasses the essential information required for the agent to make informed decisions. In our implementation, the state space includes the positions of all balls on the table. We had the option to represent the positions as continuous or discrete, however, we decided to use discrete (x, y) coordinates as this allowed us to use reinforcement learning techniques that we have used previously. Specifically, the range of where the ball's centers could be on the table was from a minimum coordinate value of $(95, 95)$ and a maximum coordinate of $(1140, 618)$. This range of values considers that

the ball's centers will not physically touch the edges unless contacting a pocket (see Fig. 2).

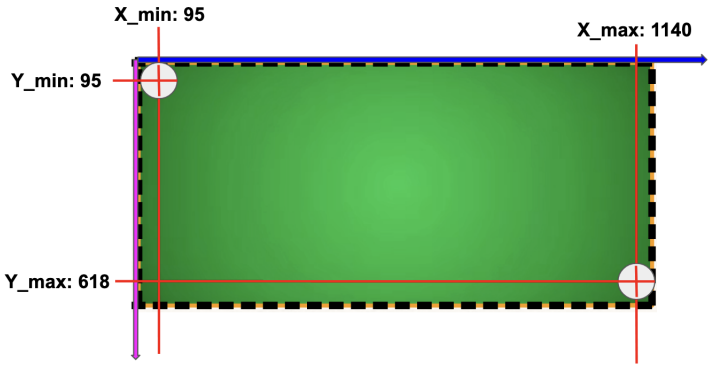

Fig. 2. Minimum and maximum ball positions along x-y axes.

2) Environment Step: As the balls move from one position to another after impact, their positions change continuously. However, once the balls stop rolling, we **snap** their final position to the nearest integer values for x and y. Therefore, a step in the billiards game environment begins with an initial integer x-y position for all 16 balls. After an agents acts upon the cue ball with a power-angle action pair, the cue ball and other balls may take motion. Once all balls have a sum velocity very close to zero, then their non-integer positions are snapped to the nearest integer x-y values on table and returned as s'. This discrete state-space provides the agent with a comprehensive view of the game's current status, enabling it to learn and devise strategies accordingly, while also allowing us to take advantage of discrete space reinforcement learning techniques.

3) Action Space: The action space represents the set of possible actions the agent can take in a given state, which, for our custom billiards environment, consists of two components: the cue ball's shooting angle (θ) and the force (F) applied to the cue ball. The shooting angle ranges from 0 to 359 degrees and the force is bounded within a predetermined range, which we chose as $[0, 20000]$ newtons. We chose to discretize the range of possible shooting angles by 719 different values, which then increments the angles by 0.5 degrees from 0 to 359 degrees. As for the possible choices of forces to apply to the cue ball we chose eight values: $[0, 500, 2000, 5000, 8000, 10000, 15000, 20000]$. By combining these two components, the agent can choose from a range of 5752 unique actions, allowing it to learn diverse and effective strategies for playing billiards.

4) Reward Function: The reward function plays a crucial role in guiding the agent's learning process by providing feedback on the outcomes of its actions. In our billiards game environment, we have designed a reward function that encourages the agent to pocket balls and complete the game as efficiently as possible. The reward function is composed of three main components:

1) A positive reward for pocketing a object ball, which is a positive reward of 100. This encourages the agent to pocket balls early in the game.
2) A negative reward of -100 for pocketing the cue ball, discouraging the agent from making inaccurate shots.
3) A small negative reward for each shot taken of -1, incentivizing the agent to complete the game with as few shots as possible.

B. Terminal States

There are two terminal states that we experimented with during development: 1) when all object balls are potted and 2) when the cue ball is potted. However, as we tested these terminal states, it was not very clear if setting potting of the cue ball as a terminal state was very helpful. Using it as a terminal state made it so that the agent couldn't improve the value of the state-action pair that led to potting the cue due to rewards earned later, but it also made it so that the random walk the agent took did not traverse depth as much as it did breadth.

By combining these components, our reward function effectively guides the agent's learning process, helping it to develop efficient and strategic gameplay.

C. Implementation

Reinforcement learning research often relies on OpenAI Gym [6] as a go-to platform for implementing environments and training agents. However, when designing a custom billiards game environment in Python, we chose to forgo the support of OpenAI Gym due to several significant considerations.

- **Computational resources and data efficiency:** Gym requires specific software dependencies and often lacks fine-grained control over the data collection process, potentially leading to inefficient data usage. For example, Gym's built-in data collection techniques might generate more data than necessary for our billiards environment, consuming valuable computational resources. As we had limited resources and a large state-action space, we designed a custom data-collection process for building a Q-table, which allowed us to optimize resource usage.
- **Overemphasis on benchmarks:** OpenAI Gym's focus on standardized benchmarks may inadvertently discourage researchers from exploring novel problems or less-common domains. By creating our own environment, we hope to encourage more diverse research and promote innovation beyond popular benchmarks.

It's important to note that these arguments do not undermine the value and usefulness of OpenAI Gym. It remains a powerful tool for reinforcement learning research and development. The platform offers ease of use, community support, and access to a wide range of pre-built environments, which make it highly beneficial for many users.

The custom billiards game environment was designed in Python using the PyMunk package for simulating the physics (collisions, momentum, elasticity) and the Pygame package for displaying the game objects as well as interacting with the game. Both PyMunk and Pygame are well-established

packages that offer the flexibility and efficiency we sought for our project.

III. Q-Learning and Deep Q-Learning

Q-learning and Deep Q-learning are two popular reinforcement learning algorithms that enable agents to learn optimal policies for decision-making in various environments. In this section, we provide an overview of these algorithms, including their respective equations, and discuss their applicability to our custom billiards game environment.

A. Q-learning

Q-learning is a model-free, value-based reinforcement learning algorithm that uses a Q-table to estimate the expected cumulative reward for each state-action pair. The Q-table is updated iteratively using the Bellman equation, which incorporates immediate rewards and discounted future rewards. The primary advantage of Q-learning is its simplicity, making it suitable for problems with small state and action spaces.

The Q-learning update equation is as follows:

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

where α is the learning rate, γ is the discount factor, s_t and a_t are the current state and action, and r_{t+1} is the immediate reward received after taking action a_t in state s_t.

However, Q-learning becomes computationally infeasible for problems with large or continuous state and action spaces, as the Q-table grows exponentially in size. This limitation makes it challenging to apply Q-learning to the custom billiards environment we aim to use, which has a large state space due to large discretization of continuous positions for all billiards balls.

B. Deep Q-learning

Deep Q-learning addresses the scalability issues of Q-learning by approximating the Q-function using a neural network, known as the Q-network. Instead of storing values in a Q-table, the Q-network learns to estimate Q-values from input states. This approximation allows Deep Q-learning to handle problems with high-dimensional state and action spaces more efficiently than traditional Q-learning.

To stabilize learning, Deep Q-learning incorporates two key techniques: experience replay and target networks. Experience replay stores past experiences in a replay buffer, enabling the algorithm to learn from a diverse set of experiences. Target networks are used to stabilize the target Q-values during training by periodically updating the target Q-network with the weights of the primary Q-network.

The loss function for Deep Q-learning is given by:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2 \right] \quad (2)$$

where θ represents the parameters of the Q-network, D is the replay buffer, and θ^- denotes the parameters of the target Q-network.

Given the large state space of our custom billiards environment, Deep Q-learning may be a more suitable choice for learning an optimal policy. However, implementing Deep Q-learning requires additional considerations, such as network architecture, hyperparameter tuning, and computational resources.

Q-learning and Deep Q-learning are both powerful reinforcement learning algorithms with distinct advantages and limitations. While Q-learning is simple and effective for small-scale problems, Deep Q-learning provides a scalable solution for problems with large or continuous state and action spaces, making it a promising candidate for our custom billiards game environment. To better understand how each of these algorithms interact with the custom billiards environment, we began with implementing traditional Q-learning.

IV. Q-Learning Implementation

The Q-learning implementation was what really made this project quite a challenge. Given that the state-action space was huge, even after discretization, we had to make significant considerations to make sure our hardware would permit storage of the Q-table.

1) Custom Q-table Implementation: The q-learning implementation was designed with a custom q-table storage data structure. We utilized Python dictionaries to represent the Q-table, with a separate dictionary for each size set of object balls on the table, ranging from 0 to 15. This allowed us to efficiently store and access the Q-values corresponding to different state-action pairs.

To reduce the memory footprint of the Q-table, we first mapped the 2D positions of the balls to 1D scalars using row- and column-major order, which allowed us to represent the positions as single integers. We further reduced the size by converting the base-10 integer values to a larger base, such as hexadecimal. This compact representation helped in reducing the memory requirements for storing the Q-values.

We also introduced a discretization scheme for ball positions, angles, and power to further reduce the state and action space. Balls were made to 'snap' to the nearest discrete position on the simulated billiards table, ensuring that the start and end states for every ball were within the states specified in the configuration parameters.

To minimize the size of the string for each q-dict entry, we used delimiters to separate cue ball state, object ball states, angle, and power. For example, we used "|" as the delimiter to separate different components of the state-action pair and "," to separate individual object ball positions.

By employing these techniques, our q-learning implementation achieved a more efficient and compact representation of the Q-table, allowing us to effectively manage memory requirements and facilitate the learning process for the billiards game.

2) Custom Q-learning Implementation: Since we had to utilize a custom implementation for a q-table, it only made sense that we implement q-learning without using any premade packages. We designed a custom Q-learning algorithm, taking into account the specific requirements of the billiards

game, such as the state representation, action space, and discretization.

First, we defined the state-action space, which consists of the cue ball position, the object ball positions, the angle of the shot, and the power applied to the cue ball. The state and action spaces were discretized, ensuring that the learning algorithm could handle the continuous aspects of the game in a manageable manner.

Next, we initialized the Q-table, which is a collection of Python dictionaries, with separate dictionaries for each size set of object balls on the table, ranging from 0 to 15. This structure allowed us to efficiently store and access the Q-values corresponding to different state-action pairs.

The Q-learning algorithm was then implemented, following the standard update rule (see (1)).

During the learning process, we used an exploration-exploitation strategy, such as ϵ-greedy, to balance the exploration of new state-action pairs and the exploitation of the current knowledge of the Q-table.

Finally, we implemented a training loop that iteratively applied the Q-learning algorithm to the billiards game for a specified number of episodes. We set the training to 200000 iterations as we wanted to leave it to train while we were gone. Each time the loop ran, it simulated the environment until only the cue ball remained.

V. Deep Q-Learning Implementation

A. Deep Q Network Architecture

1) Input layer: The input layer has a size equal to twice the dimensions of the state representation of all billiards balls on the table. Since all states of balls can be represented by a single integer value, the input size is 34 and uses the ReLU activation function.

2) hidden layers: There is one hidden layer with 64 neurons and ReLU activation function.

3) output layer: The output layer has a size equal to the number of possible actions (number of discrete forces * number discrete angles) = (8 * 719), which is a size of 5752, and uses a linear activation function to represent the Q-values of each action.

Fig. 3. The deep Q-Learning Network architecture tested on billiards environment. Illustration was generated using VisualKeras.

B. Deep Q Network loss function, optimizer, and hyperparameters

1) Loss function: The Mean Squared Error (MSE) loss function is used to measure the difference between the predicted Q-values and the target Q-values.

2) Optimizer: The Adam optimizer is used for updating the weights of the network, with a learning rate of 0.001 (default value in the create_model function).

3) Hyperparameters: The given implementation primarily uses the following hyperparameters:

1) **Learning rate:** Set to 0.001 in the create-model function.
2) **Discount factor (gamma):** The discount factor was to 0.99 to guide the agent to make decisions that would result in the most longterm rewards.
3) **Exploration rate (epsilon):** This exploration rate was to 0.02, so that the agent would explore 98% of the time.
4) **Batch size:** Tested values of 10, 20, and 50. This was a hyperparameter we did not put too much emphasis on, though we would have liked to do a larger batch size if time permitted.
5) **Memory limit:** Set to 500. This is another parameter that would would have liked to increased, retrospectively considering our state-action space.

During experimentation with different values for these hyperparameters and modifying the network's architecture (e.g., the number of layers, units per layer, activation functions) in an attempt to improve the performance of the model, it ultimately was hard to discern what did and did not improve the model. We have decided this was likely impatience to verify our architecture was minimizing loss at later episodes - we wanted immediate feedback. In future work, we would recommend to try other techniques like target network, double deep Q-Network, or prioritized experience replay to enhance the learning process.

VI. Results

The design and implementation of the Q-learning and deep Q-learning algorithms on our billiards environment were made with thoughtful consideration, but there can always be improvements. Presented below are the results are each.

1) Q-Learning Results: As mentioned in IV, the Q-learning algorithm underwent 200,000 iterations. With each simulation of the game environment, the Q-table was updated, enabling the agent to learn from its experiences and gradually improve its performance in the billiards game. However, the state-action space is too vast to be fully explored within the given number of iterations. To verify the correct execution of our implementation, we set the Q-learning algorithm to replay the final state until an action yielding a reward greater than zero was taken. We employed an epsilon-greedy strategy, where the agent chose the action with the highest Q-value for a given state 50% of the time and selected a random action the other 50%. This approach allowed the agent to follow a successful sequence of actions that resulted in sinking at least one ball in a pocket during each step while also diversifying the successful action alternatives it could discover.

2) Deep Q-Learning Results: With regard to the deep Q-learning algorithm applied to our custom billiards environment, our aim was to train it for 200 iterations. However, each iteration takes approximately 30 minutes, and after about six hours of training, it appears that the agent has been making progress based on the observed loss. There will need to be further training to know if this is for sure the case. Despite

this, the deep Q-learning algorithm has been designed to successfully interact with the billiards environment, attempting to minimize the loss per iteration (see Fig. 4). We experimented with various hidden layer configurations and hyperparameters to determine which would yield the best reduction in loss over time, but no optimal combination was found.

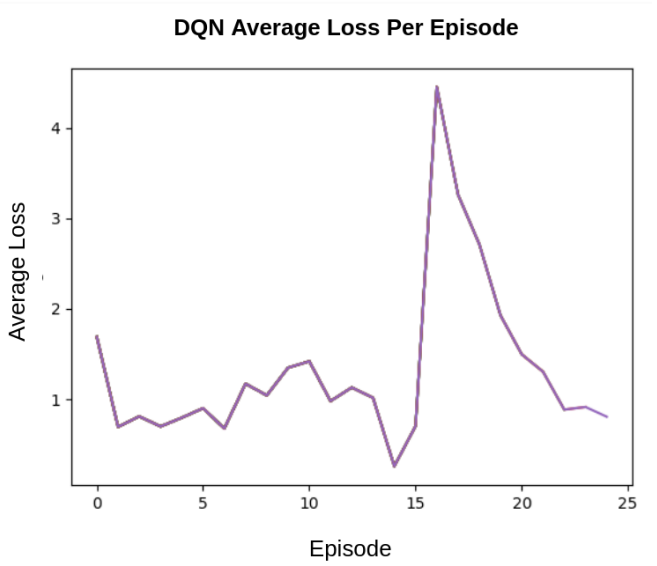

Fig. 4. Deep Q-Network Mean-Squared Error (MSE) loss for each episode in billiards game environment.

Although the deep Q-learning algorithm has been implemented correctly, the complex nature of the billiards environment could necessitate an extended training period spanning multiple days. This extended training time can be attributed to the substantial amount of simulation time required for each step within the environment.

VII. Conclusion

In this paper, we presented the design, implementation, and assessment of a custom billiards game environment, along with the development and application of a tailored Q-learning algorithm and deep Q-learning approach. Our goal was to deepen our understanding of reinforcement learning techniques and their applicability in games and simulations through the creation and deployment of these algorithms.

The Q-learning algorithm demonstrated potential for learning effective strategies within the custom billiards environment. However, the time required for the process completion was excessively long. To speed up the Q-table population process, we could consider reducing the time needed for simulating actions in the billiards game and exploring parallelization techniques for the simulation. A crucial finding in our Q-learning experimentation was our approach to storing Q-values. While many libraries use row-column major order for efficient storage and access of multi-dimensional data in computer memory, our custom method allowed us to intuitively recognize the need for such representation and independently implement it. Furthermore, our tailored approach for Q-table storage enabled us to reduce the data size per Python dictionary by categorizing Q-tables based on the number of object balls on the table. Consequently, pocketed object balls did not add sparsity to the already extensive Q-table.

Although we initiated training for a deep Q-learning approach that is showing promising results, there is still training to be done. However, we believe our implementation was done correctly. The complexities of the billiards game, the design of the neural network architecture, and potential limitations in our training methodology might have contributed to the slow training of our deep Q-learning implementation. Despite these challenges, our work has provided valuable insights into the difficulties of adapting deep Q-learning to custom game environments, paving the way for future research.

In conclusion, we hope this paper contributes to the broader understanding of reinforcement learning techniques by demonstrating the value of developing custom game environments and implementing original reinforcement learning algorithms. The challenges and insights gleaned from our work can inform future research and encourage the exploration of novel reinforcement learning applications in games and simulations, ultimately advancing the field and promoting innovation.

VIII. Future Work

A. *Raytracing for faster learning*

The work in the area of raytracing for billiards dynamics can be quite complex, even going into boundary integral frameworks to help with interpolating between a deterministic and a completely random observations of trajectory propagation [7]. Using raytracing could vastly improve a learning algorithm as it will let an agent better examine possible rollouts from a collision. While we tried to implement a simple raytracing algorithm to help predict how a collision would rollout, we ultimately had to refocus our efforts on implementing Q-learning on the custom billiards game due to time constraints. However, we did do preliminary work and brainstorming to help with this formulation (Fig. 5).

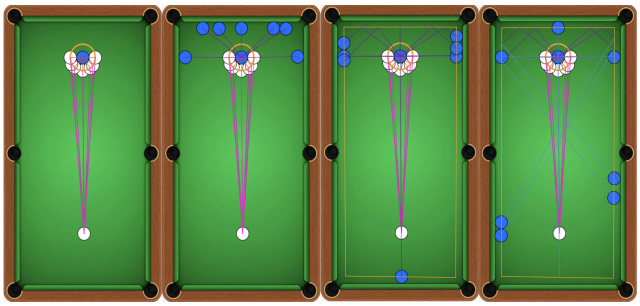

Fig. 5. Rollout of single-ball raytrace for seven different shoot angles (θ).

The first ray from a collision can be seen while playing as a human in our environment, which can be set within the *main.py* module of our code base (Fig. 6).

Acknowledgment

We would like to express our sincere gratitude to Dr. Zach Sunberg, our professor, and Jackson, our teaching assistant, for

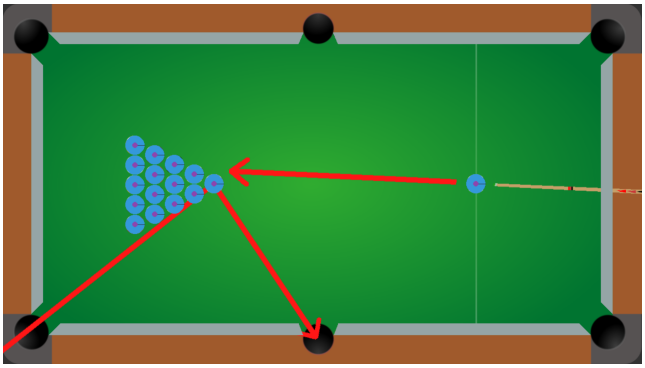

Fig. 6. A single rollout raytrace for the first object ball resulting from the current aim of the shoot angle (θ), as well as the ray from the first object ball to its nearest pocket.

their exceptional guidance during the Spring 2023 Decision-Making Under Uncertainty course at the University of Colorado - Boulder. We greatly appreciate their valuable support and assistance, which have played a significant role in the successful completion of this project.

REFERENCES

[1] R. R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Pérez-Liébana, "Deep reinforcement learning for general video game AI," *CoRR*, vol. abs/1806.02448, 2018. [Online]. Available: http://arxiv.org/abs/1806.02448

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[3] V. Mnih, K. Kavukcuoglu, D. Silver *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

[4] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[5] S. El Mekki *et al.*, "Recommender system for the billiard game," 2019.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[7] D. J. Chappell and G. Tanner, "A boundary integral formalism for stochastic ray tracing in billiards," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 24, no. 4, p. 043137, dec 2014. [Online]. Available: https://doi.org/10.10632F1.4903064