CS747 - Foundations of Intelligent and Learning Agents

Assignment 3 Report

Kaishva Chintan Shah (200020066)

Department of Electrical Engineering
Indian Institute of Technology, Bombay
November 2023

1 Introduction

This report explains the code used in a billiards game simulation. The code aims to determine the optimal direction and force to strike the white ball to achieve specific gameplay objectives. In this document, we will discuss the essential functions and their roles in the decision-making process.

2 Creating New Pseudo Holes

In some cases, the code introduces pseudo holes for gameplay considerations. These pseudo holes are created by displacing and adjusting the positions of real holes. Pseudo-holes serve a specific purpose in the game logic, allowing for additional strategic options. The original hole positions define them and can affect the decision-making process for selecting the best shot.
Here is the Python code for creating pseudo holes:

```
pseudo_holes = []
hole_radius = 24 + 1
displacement_measure = 40 + 1
hole_1 = (holes[0][0] + displacement_measure, holes[0][1] +
                                    displacement_measure)
hole_2 = (holes[1][0] + displacement_measure, holes[1][1] -
                                    displacement_measure)
hole_3 = (holes[4][0] - displacement_measure, holes[4][1] +
                                    displacement_measure)
hole_4 = (holes[5][0] - displacement_measure, holes[5][1] -
                                    displacement_measure)

hole_middle_1 = (holes[2][0], holes[2][1] + hole_radius)
hole_middle_2 = (holes[3][0], holes[3][1] - hole_radius)

pseudo_holes.append(hole_1)
pseudo_holes.append(hole_2)
pseudo_holes.append(hole_middle_1)
pseudo_holes.append(hole_middle_2)
pseudo_holes.append(hole_3)
pseudo_holes.append(hole_4)
```

In this pool game simulation, the introduced pseudo holes serve as the primary set of target points. In many instances, these pseudo-holes become the focal targets, overshadowing the holes on the table. The rationale behind introducing pseudo holes lies in resolving a significant gameplay challenge – the obstructed view of the target holes due to the table margins.
The primary issue these pseudo holes address is the frequent alignment of the ball intended for a shot with the table's edges, which would often block the line of sight to the target hole. In such cases, the player's perspective is compromised, and aiming accurately becomes problematic.
By strategically positioning the pseudo holes, this challenge is effectively mitigated. These new target points are located within the playing area, away from the margins, ensuring that each pseudo-hole remains accessible and visible from any part of the

table. This ensures that the obstructed view issue no longer plagues the gameplay experience.
In essence, creating pseudo holes enhances gameplay by providing precise and unobstructed targets, regardless of the ball's position on the table. This innovation dramatically contributes to the adaptability and strategic decision-making capabilities of the code, as it allows the simulation to navigate diverse in-game scenarios with ease.

3 Selecting the Best Ball and Hole Pair

In the billiards simulation, selecting the best ball and hole pair is essential for strategic shots. This selection is determined by the logic embedded in the `select_best_shot` function, which evaluates various factors to optimize gameplay. Here, we delve into the intuition behind this function and its role in enhancing the decision-making process.
The `select_best_shot` function is responsible for identifying the optimal combination of a ball and a hole as the primary target for a shot. It considers several key factors in its decision-making process:

- **Visibility and Line of Sight:** The function assesses the visibility of potential target holes from the current position of the white ball and candidate ball. It checks if the line of sight to a hole is unobstructed, indicating that it is a feasible target.
- **Proximity to Holes:** The proximity of the white ball and candidate balls to the available holes is considered. Closer balls are favoured as they have a higher chance of potting.
- **Obstruction Consideration:** The function accounts for possible obstructions between the white ball and the target hole. It ensures that intervening balls do not hinder the chosen shot.
- **Confidence Factor:** Confidence in the selected shot is calculated based on the degree of unobstructed visibility to the target hole. Higher confidence suggests a more favourable shot. Intuitively, it is the straightness of the shot that is calculated; a straighter shot is favourable.
- **Adaptability and Strategy:** The function promotes adaptability by selecting shots that minimize the distance between balls and holes. This aids in strategic gameplay and potentially leads to higher success rates. We have given scores to each pair of balls and holes based on all the factors discussed above; the weights for these factors were fine-tuned based on observations from experiments conducted, and the highest-scored pair is then returned.

The intuition behind the `decide_best_shot` function is to provide the simulation with the ability to make informed and strategic decisions. By assessing visibility, proximity, and potential obstructions, it selects shots with a higher probability of

success. Furthermore, by considering adaptability and strategy, the function ensures that the gameplay remains dynamic and responsive to the in-game situation.
The selected ball and hole pair are crucial, as they define the direction and force for the subsequent shot. This function dramatically contributes to the code's capability to optimize shots and navigate through complex scenarios, ultimately enhancing the overall gaming experience.

4 Shooting Angle Determination

To determine the angle at which the white ball should be shot, the `decide_white_shooting_directio` function is utilized. The goal is to find the optimal shooting direction for the white ball to pot a specific ball into a pseudo hole. This process involves several key considerations:

- **Optimal Line of Contact:** The algorithm seeks an optimal line of contact between the white ball, the target ball, and the target hole. The idea is to strike the target ball so that it moves along a path leading to the desired hole.
- **Angle Calculation:** The function calculates the angle between the target hole and the target ball, allowing the white ball to be shot along this angle. This angle is determined by considering the positions of the target hole, the target ball, and the white ball.
- **Line of Contact Calculation:** After computing the angle, the code calculates the point of contact (intersection) between the target ball and the line corresponding to the calculated angle.

Here's the Python code for the `decide_white_shooting_direction()` function:

```
def decide_white_shooting_direction(self, white_ball_pos, ball_pos,
                                    hole_pos_label, pseudo_holes):
    hole_pos = pseudo_holes[hole_pos_label]
    theta = self.angle_vector(hole_pos, ball_pos)
    point_of_contact = self.line_joining(ball_pos, theta, 2*self.
                                         ball_radius)
    if([40, 40] in pseudo_holes): # To account for the passage at
                                    the corner holes
        point_of_contact = self.line_joining(ball_pos, theta, self.
                                             ball_radius)
    return self.angle_vector( white_ball_pos, point_of_contact)
```

In this code snippet, `theta` represents the angle between the target hole and the target ball. The function `angle_vector()` computes this angle based on the positions of these points. The `line_joining()` function determines the point of contact (intersection) by projecting a line from the target ball along the calculated angle.
The intuition here is to align the white ball so that it strikes the target ball at the right angle to guide it toward the target hole. This ensures that the target ball follows a trajectory that leads it into the desired hole.

5 Determining Optimal Force for a Smart Shot

The `decide_force_smart()` function is responsible for deciding the optimal force to be applied when taking a smart shot with the white ball. The smart shot aims to pot a specific ball into a target hole. To make this decision, the function goes through a series of steps and considerations:

5.1 Optimizing Force

First, the function defines a range of force values to test within. These values are systematically varied to find the optimal force that results in potting the ball. A set of initial force values is used for this optimization. These values are chosen considering different scenarios and can be adjusted based on experience.

```
initial_force_range = np.linspace(0, 1.0, 5)
for initial_force in initial_force_range:
    seed = np.random.randint(0, 100000)
    force = self.optimize_force(white_ball_pos, ball_pos, hole_pos,
                                initial_force, direction,
                                ball_pos_dict, ball_label,
                                seed)
    if force is not None:
        return force
```

5.2 Force Optimization

The `optimize_force()` function is called to fine-tune the force required for potting the ball. It aims to iteratively adjust the force value based on the results of each shot. Make note that we check this only with the original holes and not the pseudo holes. This is because the force needs to get the ball to reach the original holes rather than the pseudo holes and get stuck there.

```
def optimize_force(self, white_ball_pos, ball_pos, hole_pos,
                                initial_force, direction,
                                ball_pos_dict, ball_label, seed):
    max_iterations = 5
    force = initial_force
    previous_shot_success = False
    previous_state = ball_pos_dict

    for _ in range(max_iterations):
        action = (direction/math.pi, force)
        next_state = self.ns.get_next_state(ball_pos_dict, action,
                                            seed)
        if self.ball_potted(next_state, ball_pos_dict):
            return force  # Return the force that results in
                                                potting the ball

        # If the shot didn't pot the ball, adjust the force for the
                                            next iteration
        if previous_shot_success:
            force += 0.04  # Increase force for fine-tuning
```

```
        else:
            force += 0.01  # Smaller step size when not making
                                                progress

        previous_shot_success = False

        if self.distance(next_state[ball_label], hole_pos) < self.
                                            distance(previous_state[
                                            ball_label], hole_pos):
            previous_shot_success = True

        previous_state = next_state

    return None  # No force value resulted in potting the ball
```

The `optimize_force()` function iterates multiple times, simulating a shot by adjusting the force. If the shot results in potting the ball, the function returns the force value that achieves this. If no force value within the specified range leads to potting the ball, the function returns `None`.

5.3 Checking If a Ball Is Potted: `self.ball_potted()`

The `self.ball_potted()` function is crucial for determining if a ball is successfully potted during a simulated shot. It checks whether the number of colour balls (excluding the white ball and label 0) in the current state is one less than the number of colour balls in the next state. This indicates that a ball has been potted.

```
def ball_potted(self, next_state, ball_pos):
    current_num_colour_balls = sum(1 for label in ball_pos if label
                                        != 'white' and label != 0)
    next_num_colour_balls = sum(1 for label in next_state if label
                                       != 'white' and label != 0)
    return current_num_colour_balls - next_num_colour_balls >= 1
```

The function compares the count of colour balls in the current state to the count in the next state. If it finds that the count has decreased by 1, it returns `True`, indicating that a ball has been potted; otherwise, it returns `False`.

5.4 Distance Calculation: `self.distance()`

The `self.distance()` function calculates the Euclidean distance between two points. It's utilized in various parts of the code to measure the distance between objects on the pool table.

```
def distance(self, point1, point2):
    return np.linalg.norm(np.array(point2) - np.array(point1))
```

The function takes two points as input and returns the Euclidean distance between them. This distance calculation is vital for assessing the proximity of balls to each other, the white ball, and the target holes.
By using these functions, the code effectively checks whether a ball has been potted and calculates distances, which are essential for making informed decisions when

determining the optimal force for smart shots. The `self.ball_potted()` function is particularly valuable in verifying the success of a shot.

5.5 Determining Force for Standard Shots: `self.decide_force()`

The `self.decide_force()` function determines the force applied to the white ball in standard shots. This function is crucial in scenarios where the optimization process doesn't yield a force value.

```
def decide_force(self, white_ball_pos, ball_pos, best_hole_label,
                                    holes, confidence, straight_shot)
                                    :
    hole_pos = holes[best_hole_label]
    if self.distance(ball_pos, hole_pos) <= 30:
        return 0.8
    theta = self.angle_vector(hole_pos, ball_pos)
    point_of_contact = self.line_joining(ball_pos, theta, 2 * self.
                                    ball_radius)
    direction_of_white_ball = self.angle_vector(white_ball_pos,
                                    point_of_contact)
    distance_white_colour = np.sqrt((white_ball_pos[0] - ball_pos[0
                                    ]) ** 2 + (white_ball_pos[1]
                                    - ball_pos[1]) ** 2)
    distance_white_colour /= np.sqrt(960 ** 2 + 460 ** 2)
    if straight_shot:
        return 0.5 * distance_white_colour + confidence * 0.3
    return 0.5 * distance_white_colour + confidence * 0.3
```

The `self.decide_force()` function accepts various parameters including the position of the white ball (`white_ball_pos`), positions of other balls (`ball_pos`), the label of the best hole (`best_hole_label`), an array of hole positions (`holes`), confidence value (`confidence`), and a flag for straight shots (`straight_shot`).
The function evaluates the distance between the target hole and the current ball, adjusting the force accordingly. It uses the angle between the hole and ball (`theta`) to calculate the point of contact, considering the radius of the ball. It then calculates the direction of the white ball at this point of contact.
The force is determined based on the distance between the white ball and the current ball and is influenced by the confidence factor. If it's a straight shot, a specific force value is returned. Otherwise, the calculated force is returned.
This function is essential for providing a force value when the optimization process doesn't yield a result, ensuring that shots are made with appropriate power.
By understanding the `self.decide_force()` function, we gain insights into how force is determined for standard shots in various game scenarios.

5.6 Intuition

The underlying principle of this approach is to methodically explore a spectrum of force values while continuously refining them to pinpoint the ideal force for pocketing the ball. This adaptive strategy dynamically adjusts the force based on the outcome of each shot, thus optimizing the probability of accurately sinking the target ball

into the intended pocket. However, in the rare event that the optimization process fails to yield an appropriate force value, we rely on our meticulously calibrated `decide_force` function as a fallback measure.
This method allows for flexibility and adaptability in choosing the force, as different shots may require varying levels of force. The optimization process considers factors such as the direction of the shot and seed randomness, ensuring a more robust decision on the force to apply.

6 Transitioning from Pseudo Holes to Original Holes

In this section, we explore how the code adapts its strategy when the `select_best_shot` function does not yield any viable ball and hole pair within the set of pseudo holes. This scenario often indicates that the target ball has moved into the confined region between the pseudo holes and the original holes on the billiard table.

6.1 Reverting to Original Holes

When the code encounters such a situation, it intelligently switches its approach to targeting the original holes on the billiard table. This transition is crucial for achieving a higher degree of accuracy in potting the ball into one of the original pockets.

6.2 Optimal Alignment

The decision to revert to the original holes is based on recognising that the limited space between the pseudo holes and the table's edges demands precise alignment for successful potting. By focusing on the centre of the target ball and aiming for the original pockets, the code optimizes its strategy to adapt to varying spatial conditions.

6.3 Adaptability of the Code

This change in targeting strategy underscores the code's adaptability to diverse scenarios, ensuring it employs specific strategies tailored to the spatial dynamics of the billiard table.

7 Perturbing the Shot

The code employs a perturbation strategy to compensate for potential errors introduced during simulation. This strategy adds a controlled element of randomness to the shot's angle, aiming to counteract any deviations introduced by the simulation process.

7.1 Intuition Behind Perturbation

The perturbation strategy is rooted in the understanding that the simulation process may introduce inherent errors, potentially affecting the shot's accuracy. To mitigate this, the code introduces an intentionally applied error to the shot's direction.
The intuition is that by adding a perturbation to the shooting direction, we aim to create an error with an opposite sign to that which the simulation may introduce. This counteractive perturbation is designed to increase the likelihood of achieving the intended shot outcome, even in simulated errors.
In essence, perturbing the shot is a preemptive measure to compensate for potential discrepancies, enhancing the code's resilience and adaptability in dynamic billiard scenarios.

7.2 Implementation

The magnitude of the perturbation is directly proportional to the force applied to the shot. Higher force results in larger perturbation, providing a greater degree of error compensation. The code ensures that perturbation remains within reasonable bounds to maintain precision and control.
In summary, the perturbation strategy proactively responds to potential errors, aligning with the code's adaptability and strategic finesse.