A Modifiable and Realistic Football Simulation: Using Action-defined States with Normalized Categories in Q-learning

Konuralp Sarisozen
Computer Science, NYUAD
ks6299@nyu.edu

Advised by: Kenneth Perlin, Yasir Zaki

ABSTRACT

This report explains Konuralp Sarisozen's Capstone Project, creating a computationally inexpensive and realistic 2D soccer simulation that reflects real-life tactics and decision-making. A football simulation should include user tactics and rewards harmonically while giving agency to the players themselves in decision-making, requiring a strategy of reinforcement learning, such as Q-learning. This simulation, for the Q-learning-based decision-making process of the agent who currently possesses the ball, defines his state in relation to the predetermined rationality of taking each action based on various environmental factors -**not the Q-values**-, while normalizing these rationality values to smooth the process of categorizing the states and adjusting the number of states to ensure easiness in compatibility to future updates. These rationality scores just inform the state representation, making the state more expressive; they do not dictate which action the agent takes. Here is the link for the simulation video: Simulation Video

KEYWORDS

Q-learning, reinforcement learning, artificial intelligence, football, soccer, computational geometry

Reference Format:
Konuralp Sarisozen. 2025. A Modifiable and Realistic Football Simulation: Using Action-defined States with Normalized Categories in Q-learning. In *NYUAD Capstone Seminar Reports, Spring 2025, Abu Dhabi, UAE.* 16 pages.

1 INTRODUCTION

This project is important because it can provide insights into the current gaming industry of football and potential future simulations used in real-life football. The simulation tries to provide an efficient way of designing the Q-tables not only for harmonizing varied tactical weights and success/failure based rewards but also for providing a modifiable and compatible approach for increasing the number of states and arranging states for the new ideas and updates. Q-learning has been chosen because it requires minimum levels of computational power, yet it can be realistic if the environment is designed well enough with efficient categorization of the states.

The user's tactical inputs affect the team he is controlling because tactical weights are included in the evaluation of the agents' actions, and the players have the agency to explore actions with a rate and learn from their actions to make the best decisions for themselves, affected both by the tactics and success/failure of their actions. Besides, the rewarding system also includes possible future success according to the Q-values of the new state that occurs as a result of the current action, to account for chain reactions. An evaluation of an action is based on tactics, immediate success/failure, and future success if the current action ends up as a valid next state.

Tactics, however, should affect the evaluation of the actions differently for different states based on the relationship between the state and all the actions. For instance, a user can give a tactic, "Shoot More" to encourage players to shoot. In real life, this tactic would not add a constant positive weight to the decided action's -in this case, shooting- learning function for all possible situations where a player can shoot; rather, its encouragement level would vary in different situations. It would encourage shooting more when a player is more likely to shoot, and it would encourage less when a player is less likely to shoot, but it would still add up weight in rewarding all shootings, just with varying values. Therefore, a football simulation should also adjust different tactical weights for different states based on these states' relationship to the actions themselves.

At this point, this project decides to define its states based on the predetermined rationality of taking the actions in that given moment in the environment. These rationality scores of actions (shooting, dribbling, passing) included in the state definitions are different than the Q-values of the actions. They are defined before simulation, and they do not change. They do not dictate which action the agent takes. Instead, they inform the state representation, making the state more expressive. The Q-learning agent still relies on its Q-values to decide which action to take, based on trial-and-error and observed feedback. These rationality scores within the definition of the states only help arranging the tactical weights and success/failure weights to influence rewards in the Q-learning function. These rationality scores are normalized between 0 and 1 so that it becomes easier to categorize and modify states and corresponding tactical and success/failure weights that would be used in the actions' Q-learning function.

For passing, a second layer of a Q-table is designed for an agent to decide how exactly to pass because certain tactics affect not only the decision to shoot, dribble, or pass but also how to pass if a passing decision is made from the main layer of the Q-table.

In addition to the in-game learning, the project is enhanced with simulation training, meaning that q-values are not initialized as 0 when a simulation starts, but initialized with trained values to enhance realism. Tactics and impacts of the tactics on the learning is also trained and evaluated.

Each decision and action is coded to reflect real-life soccer dynamics with a certain level of randomization of variables to enhance smooth simulation.

The movements of the other players who do not have the ball are based on rule-based AI algorithms developed, and they do not include reinforcement learning. Basing general movement decisions on reinforcement learning is out of the scope of this prototype, but it is considered for future enhancements.

This report does not explain every detail of the project; rather, it delves into the most important parts.

2 RELATED WORK

Sequential decision-making has long been studied in computer science. Bellman's Dynamic Programming [2] is foundational in the dynamic decision-making process to make the optimal decision. Bellman equation is the basis of Q-learning and its formula. Therefore, everything in this report is built upon the ideas and formula of Bellman. However, this paper develops various unique approaches to Q-learning with respect to its methodology and implementation to adjust it for an in-game learning function for the football simulation.

Arslan and Yuksel [1] present decentralized Q-learning algorithms for stochastic games and analyze their convergence properties for weakly acyclic games, which include team problems as an important special case. The algorithms are decentralized, meaning each decision maker only has access to its own decisions, costs, and state transitions, without knowledge of other decision makers. The paper introduces two main algorithms: one that converges to equilibrium policies in games that are weakly acyclic under strict best replies, and another that converges in the broader class of games weakly acyclic under strict better replies. The key idea is to use exploration phases where decision-makers maintain constant policies for extended periods, allowing them to accurately learn Q-factors in a temporarily stationary environment. Their approach to decentralization is inspirational, and this project also uses a similar approach by decentralizing the two teams in Q-learning and evaluating each team separately.

Leslie, David, and Collins [4] discuss the challenges of applying reinforcement learning to multi-agent scenarios, specifically focusing on the iterated normal form game. The authors examine the behavior of value-based learning agents in this context and show that such agents generally cannot play at a Nash equilibrium, although they can reach a Nash distribution using smooth best responses. The paper introduces a new algorithm called individual Q-learning and analyzes its asymptotic behavior using stochastic approximation. Key findings include the convergence of strategies to Nash distribution in 2-player zero-sum games and 2-player partnership games.

These works have mostly been based on optimizing the decision-making process; however, this paper demonstrates a simulation-based game, and its aim is not to completely optimize the decisions, rather, to optimize the decisions based on the integration of tactics into the actions. Besides, the exploration rate in this paper is not just used to explore possible new best actions but also to make mistakes to reflect real life. Therefore, it is higher than usual. The Q-learning in this simulation, therefore, is an adjusted version to establish a proper simulation environment.

Another paper [3] explains the evolution of Q-learning and its different types. Even though the football simulation presented in this paper modifies conventional Q-learning, Q-learning's developments displayed in the article throughout the years provided insights into the modifications made to Q-learning to establish a proper simulation environment.

Another paper [5] investigates the use of Q-learning, a reinforcement learning technique, to improve the behavior of game bots in first-person shooter (FPS) games. The authors developed a simplified simulation of Counter-Strike to conduct experiments on three key aspects: learning to fight, learning to plant bombs, and learning for deployment. They argue that current game AI often relies on pre-programmed finite-state machines with hard-coded rules, resulting in predictable behavior. The researchers propose using Q-learning as a simple, efficient, and online learning algorithm to dynamically evolve bot behavior. Their experiments focus on how well bots can learn basic behaviors like fighting and bomb planting, as well as how knowledge learned from abstract models can be applied to more detailed world models. This paper actually reflects a more similar environment compared to the simulation in this game. Its use of Q-learning encouraged the author of this paper to use Q-learning in this paper's simulation.

Rein and Memmert's [6] research explores big data applications in soccer, particularly for tactical analysis. By integrating these insights, the simulation's AI could simulate realistic and strategic behaviors based on large datasets of real-game tactics and outcomes. This could be crucial for developing an AI that not only reacts in real-time to player inputs but also adapts its strategy based on prevailing game conditions and opponent tactics. The article provides insights for the Capstone Project 2. For this prototype, no real-life data is directly implemented to initialize the Q-table. However, for future work, real-life data can be used in the simulation as well.

3 METHODOLOGY AND IMPLEMENTATION

3.1 Decision-making For The Player Who Possesses The Ball

3.1.1 Defining the States and Initializing The Q-tables Per Team. For the player possessing the ball, there are three possible actions to take: shoot, dribble, and pass. After making this decision, further layers of Q-tables can be added to handle how to execute each action. For the main layer with these three actions, we define our states such that each state has four components that can take different values: player position, shooting rationality category, dribbling rationality category, and passing rationality category. To integrate and differentiate tactical weights in the rewarding system of updating the Q-values, the project chooses to define its states by the predetermined score of rationality of each action so that when a decision is made based on Q-values, logical weights for the tactics that reflect the state-action relationship can be included in the Q-learning function to update the Q-values. These rationality scores have no direct impact on decision-making. It is just part of a component of the states to assign better values for the learning function per each action, as shown in section 3.1.2.

Component 1, Position of The Player: The simulation implements Q-learning for the decision-making of the player who currently possesses the ball; therefore, Q-learning is used for the team in possession at one moment. Therefore, there is a need to approach to the two teams as two different agents, each of whom has eleven sub-agents who are the players on the pitch. Because we are interested in one player, the one who has the ball, at one moment, that player can be representative of the team as an agent. Therefore, the agent is the team itself, and the player of the team having the ball represents the whole team as an agent per moment in the Q-table. This requires each state to include information about the player, his position. In football, players are defined by their positions, and the situations they would face in a game are also dependent on the positions. For simplicity in this prototype simulation, we use four possible positions (goalkeeper, defender, midfielder, attacker) in our states as one component.

Component 2, Shooting Rationality Score Category: An algorithm is developed to calculate a normalized shooting rationality score between 0 and 1 for any possible situation for the player, 0 suggesting that it is perfectly irrational to shoot and 1 suggesting that it is perfectly rational to shoot. Shooting rationality depends on three factors:

- The angle between the goal line vector and goal-to-player vector: It would be more rational to shoot if this angle is $\pi/2$ because it means that the agent is right in front of the goal. Deviations from $\pi/2$ would mean deviations from the frontal shooting position.
- The distance from the goal: The more the distance, the less likely a shoot would succeed.
- The angle between the player-to-goal vector and player-to-closest-defender vector: If this angle is π, it would mean that the closest defender to the shooter is right behind the shooter with respect to the goal's position.

Therefore, the closest opponent player to the shooter is in the opposite direction of shooting when this angle equals π. Deviations from π would decrease shooting rationality. Below is how these three factors are calculated and summed up as a normalized shooting rationality score:

(1) **Define the Vectors:**

$$\mathbf{H} = \begin{bmatrix} \text{goal.x1} - \text{goal.xCenter} \\ \text{goal.y1} - \text{goal.yCenter} \end{bmatrix}, \qquad \text{(Goal line vector)}$$

$$\mathbf{G} = \begin{bmatrix} \text{goal.xCenter} - \text{player.x} \\ \text{goal.yCenter} - \text{player.y} \end{bmatrix}, \qquad \text{(Player-to-goal vector)}$$

$$\mathbf{G}_{\text{inverse}} = \begin{bmatrix} \text{player.x} - \text{goal.xCenter} \\ \text{player.y} - \text{goal.yCenter} \end{bmatrix}, \qquad \text{(Goal-to-player vector)}$$

$$\mathbf{O} = \begin{bmatrix} \text{closestDefender.x} - \text{player.x} \\ \text{closestDefender.y} - \text{player.y} \end{bmatrix}. \qquad \text{(Player-to-defender vector)}$$

(2) **Calculate the Angle Score:**

$$\cos\theta_G = \frac{\mathbf{H} \cdot \mathbf{G}_{\text{inverse}}}{\|\mathbf{H}\| \cdot \|\mathbf{G}_{\text{inverse}}\|}, \quad S_{\text{angle}} = 1 - |\cos\theta_G|,$$

(3) **Calculate the Distance Score:**

$$S_{\text{distance}} = \max\left(0, 1 - \frac{\|\mathbf{G}\|}{R}\right),$$

where R is the maximum allowed shooting distance. (35 meters for our simulation)

(4) **Calculate the Defender Angle Score:**

$$\cos\theta_O = \frac{\mathbf{G} \cdot \mathbf{O}}{\|\mathbf{G}\| \cdot \|\mathbf{O}\|}, \quad S_{\text{defangle}} = \frac{\arccos(\cos\theta_O)}{\pi},$$

(5) **Combine the Scores:**

$$S_{\text{score}} = \text{ifAllowedDistance} \cdot (W_{\text{angle}} \cdot S_{\text{angle}} + W_{\text{distance}} \cdot S_{\text{distance}} + W_{\text{defangle}} \cdot S_{\text{defangle}}),$$

where:
- ifAllowedDistance = 1 if $\|\mathbf{G}\| <$ R, otherwise 0,
- $W_{\text{angle}}, W_{\text{distance}}, W_{\text{defangle}}$ are weights summing up to 1, assigned to each normalized sub-scores $S_{\text{angle}}, S_{\text{distance}}, S_{\text{defangle}}$

This calculation, per state, gives us a shooting rationality score: S_{score}. This normalized score is categorized into 6: Category 0: 0, Category 1: (0,0.2], Category 2: (0.2,0.4], Category 3: (0.4, 0.6], Category 4: (0.6, 0.8], Category 5: (0.8, 1]. These categories- 0,1,2,4,5- are six possible values for our state's second component.

Component 3, Dribbling Rationality Score: Similarly, we calculate a normalized score for dribbling rationality. The factors affecting the score are:

- Distance to the closest opponent: The more the distance, the easier to dribble.
- The availability of the closest five teammates (the average distance of the closest five teammates to the current agent from the closest opponent to each of them): The lower the availability, the higher the incentive to dribble as a substitute for passing
- Angle Between player-to-goal and player-to-closest-defender vectors: π would mean the closest opponent is right behind the dribbler compared to the opposition's goal's position. Deviations from π would discourage dribbling because dribbling toward the goal is more valuable, and π is when the closest defender is in the opposite direction compared to the more desired dribbling direction. Here is the brief implementation of the calculation:

(1) **Distance to the Closest Opponent:**

$$\mathbf{O} = \begin{bmatrix} \text{closestDefender.x} - \text{player.x} \\ \text{closestDefender.y} - \text{player.y} \end{bmatrix},$$

$$D_{\text{opponent}} = \min\left(1, \frac{\|\mathbf{O}\|}{R_{\text{opponent}}}\right).$$

where R_{opponent} is a threshold minimum distance that suggests a perfectly easy dribbling, used to normalize the distance effect.

(2) **Teammate Availability:**

$$D_{\text{avgTeammateAvailability}} = \frac{1}{5}\sum_{i=1}^{5} \|\mathbf{teammate}_i - \mathbf{closestOpponent}_i\|,$$

$$D_{\text{teammates}} = \max\left(0, 1 - \frac{D_{\text{avgTeammateAvailability}}}{R_{\text{teammates}}}\right).$$

where $R_{\text{teammates}}$ is a threshold average distance suggesting that teammates are too available to dribble and not to pass.

(3) **Angle Between Player-to-Goal and Player-to-Closest-Defender:**

$$\mathbf{G} = \begin{bmatrix} \text{goal.xCenter} - \text{player.x} \\ \text{goal.yCenter} - \text{player.y} \end{bmatrix},$$

$$\cos\theta = \frac{\mathbf{G} \cdot \mathbf{O}}{\|\mathbf{G}\| \cdot \|\mathbf{O}\|}, \quad D_{\text{angle}} = \frac{\arccos(\cos\theta)}{\pi}.$$

(4) **Combine the Scores:**

$$\text{DribbleScore} = W_{\text{opponent}} \cdot D_{\text{opponent}} + W_{\text{teammates}} \cdot D_{\text{teammates}} + W_{\text{angle}} \cdot D_{\text{angle}}.$$

where $W_{\text{opponent}}, W_{\text{teammates}} W_{\text{angle}}$ are weights summing to 1 to combine these scores and maintain normalization.

The calculated dribbling rationality score is categorized into 5: Category 1: (0,0.2], Category 2: (0.2,0.4], Category 3: (0.4, 0.6], Category 4: (0.6, 0.8], Category 5: (0.8, 1]. These categories-1,2,3,4,5- are five possible values for our state's third component.

Component 4, Passing Rationality Score: Similarly, teammate availability of all the teammates and the agent's average distance to all the teammates are the two factors affecting passing rationality score:

(1) **Teammate Availability:**

$$D_{\text{avg}} = \frac{1}{N}\sum_{i=1}^{N}\|\mathbf{teammate}_i - \mathbf{closestOpponent}_i\|,$$

$$P_{\text{availability}} = \min\left(1, \frac{D_{\text{avg}}}{R_{\text{pass}}}\right),$$

where R_{pass} is the normalization radius, N is the number of teammates, and $\mathbf{closestOpponent}_i$ is the closest opponent to $\mathbf{teammate}_i$.

(2) **Player-to-Teammate Distance:**

$$D_{\text{player-to-teammates}} = \frac{1}{N}\sum_{i=1}^{N}\|\mathbf{player} - \mathbf{teammate}_i\|,$$

$$P_{\text{player}} = \max\left(0, 1 - \frac{D_{\text{player-to-teammates}}}{R_{\text{player}}}\right),$$

where R_{player} is the maximum allowed distance for normalization.

(3) **Combine the Scores:**

$$\text{PassingScore} = w_{\text{availability}} \cdot P_{\text{availability}} + w_{\text{player}} \cdot P_{\text{player}},$$

where $w_{\text{availability}}$ and w_{player} are the weights assigned to teammate availability and player-to-teammate distance, respectively.

This passing rationality score is categorized into 2:Category 1: (0,0.2], Category 2: (0.2,0.4], Category 3: (0.4, 0.6], Category 4: (0.6, 0.8], Category 5: (0.8, 1]. These categories-1,2,3,4,5- are five possible values for our state's fourth component.

Combining all four components together, an example state would be (mid,1,1,2), where the player having the ball is a midfielder, his shooting rationality category, dribbling rationality category, and passing rationality categories are 1,1,2 respectively. With this approach, we have 451 states, with minor adjustments. For simplicity, below, Table 1 is shown as an example of a Q-table in which there are 37 states (less categories (0,1,2)). In the project, we use a table consisting of 451 states for enhanced realism. This, in fact, shows that our methodology allows to change the scope of the project easily.

The Goalkeeper state is hard-coded, and "0" Q-values indicate that the action is not permitted by the simulation environment for the given state. For instance, there will not be shooting for distances representing more than 35 meters in our simulation; hence, that action is not initialized for the corresponding state.

Table 1: Q-table First Layer

State	Shoot	Dribble	Pass
def,1,1,1	Q(s,a)	Q(s,a)	Q(s,a)
def,1,1,2	Q(s,a)	Q(s,a)	Q(s,a)
def,1,2,1	Q(s,a)	Q(s,a)	Q(s,a)
def,1,2,2	Q(s,a)	Q(s,a)	Q(s,a)
def,2,1,1	Q(s,a)	Q(s,a)	Q(s,a)
def,2,1,2	Q(s,a)	Q(s,a)	Q(s,a)
def,2,2,1	Q(s,a)	Q(s,a)	Q(s,a)
def,2,2,2	Q(s,a)	Q(s,a)	Q(s,a)
mid,1,1,1	Q(s,a)	Q(s,a)	Q(s,a)
mid,1,1,2	Q(s,a)	Q(s,a)	Q(s,a)
mid,1,2,1	Q(s,a)	Q(s,a)	Q(s,a)
mid,1,2,2	Q(s,a)	Q(s,a)	Q(s,a)
mid,2,1,1	Q(s,a)	Q(s,a)	Q(s,a)
mid,2,1,2	Q(s,a)	Q(s,a)	Q(s,a)
mid,2,2,1	Q(s,a)	Q(s,a)	Q(s,a)
mid,2,2,2	Q(s,a)	Q(s,a)	Q(s,a)
att,1,1,1	Q(s,a)	Q(s,a)	Q(s,a)
att,1,1,2	Q(s,a)	Q(s,a)	Q(s,a)
att,1,2,1	Q(s,a)	Q(s,a)	Q(s,a)
att,1,2,2	Q(s,a)	Q(s,a)	Q(s,a)
att,2,1,1	Q(s,a)	Q(s,a)	Q(s,a)
att,2,1,2	Q(s,a)	Q(s,a)	Q(s,a)
att,2,2,1	Q(s,a)	Q(s,a)	Q(s,a)
att,2,2,2	Q(s,a)	Q(s,a)	Q(s,a)
def,0,1,1	0	Q(s,a)	Q(s,a)
def,0,1,2	0	Q(s,a)	Q(s,a)
def,0,2,1	0	Q(s,a)	Q(s,a)
def,0,2,2	0	Q(s,a)	Q(s,a)
mid,0,1,1	0	Q(s,a)	Q(s,a)
mid,0,1,2	0	Q(s,a)	Q(s,a)
mid,0,2,1	0	Q(s,a)	Q(s,a)
mid,0,2,2	0	Q(s,a)	Q(s,a)
att,0,1,1	0	Q(s,a)	Q(s,a)
att,0,1,2	0	Q(s,a)	Q(s,a)
att,0,2,1	0	Q(s,a)	Q(s,a)
att,0,2,2	0	Q(s,a)	Q(s,a)
gk,0,0,2	0	0	Q(s,a)

One of the tactics chosen by the user is the idea(possession, balance, counter). In cases of passing, possession and counter would encourage different types of passing. Possession would encourage a less direct pass to the teammates' feet, whereas counter would encourage a more direct, forward, pass to the attacking direction and to the space where the target player would like to run. Therefore, passing is categorized based on the factors of directness and to-feet/to-space, and because the "idea" tactic impacts this decision, a second layer of Q-table is created for agents to decide how to pass, if passing

Table 2: Q-table Second Layer

State	PTFL	PTSL	PTFD	PTSD
def,1,1,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,1,1,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,1,2,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,1,2,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,2,1,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,2,2,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,2,1,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,2,2,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,1,1,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,1,1,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,1,2,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,1,2,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,2,1,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,2,2,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,2,1,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,2,2,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,1,1,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,1,1,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,1,2,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,1,2,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,2,1,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,2,2,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,2,1,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,2,2,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,0,1,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,0,1,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,0,2,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
def,0,2,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,0,1,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,0,1,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,0,2,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
mid,0,2,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,0,1,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,0,1,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,0,2,1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
att,0,2,2	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
gk,0,0,2	0	Q(s,a)	0	Q(s,a)

decision is made in the first layer. The actions are pass to feet less direct (PTFL), pass to space less direct(PTSL), pass to feet direct (PTFD), pass to space direct(PTSD). Below is also a simplified example with 37 states; the actual project uses a table of 451 states.

The advantage of having states with components having categories of normalized values is that it is easy to modify the states. We already have the normalized values and their calculations, and varying categorization will categorize the states even into smaller chunks. There are infinitely many real numbers between 0 and 1, and this enables the simulation to cover infinitely many states with certain partitions. Categorizing into a larger number of groups would easily increase the number of states and state differentiation, boosting the realism and quality of the simulation but increasing the complexity. For instance, we could easily categorize rationality components as 1: [0,0.1), 2: [0.1,0.2), 3: [0.2,0.3), 4: (0.3,0.4] ... and so on up to 1 to differentiate the situations more and perhaps differentiate the tactics more too.

3.1.2 Making A Decision and Updating the Q-tables.

The agent chooses the action with the highest Q-value with a rate of $1 - explorationRate$ and one of the other actions with equal probability between them with a rate of $explorationRate$. The $explorationRate$ is affected by the "creativity" tactic too. A higher "creativity" chosen by the user increases the $explorationRate$. The default exploration rate chosen for the simulation's current tables is 0.4 to observe the learning process better for our prototype. A considerably high rate is chosen to test in-game learning.

The Q-learning function that updates the tables takes into account the tactical weight per current state-action pair and success/failure weight in its immediate reward calculation. It also considers the future rewards and chain reaction by considering the next state's Q-values in accordance with the Q-learning methodology and Bellman equation, but also slightly different from them. Instead of focusing solely on the maximum Q-value in the next state as in conventional Q-learning, we use the average of Q-values weighted by their probabilities with respect to the exploration rate for the next state's impact to reflect real life more instead of focusing on the benefits of more exploitation of the best decision. Taking the weighted average ensures the consideration of exploration and potential mistakes of the agents. Exploration, in certain situations, reflects making mistakes. Considering these for the impact of the next state on the current state's Q-values can be helpful to reflect the real-life occasions better. Here is the algorithm for updating both of the tables per team:

(1) **Extract Q-values for the Next State:**

$$\text{nextQValues} = \{\text{Q-values of all actions from the next state in } qTable\}$$

(2) **Find the Maximum Q-value and Other Values for The Next State:**

$$Q_{\text{MaxNext}} = \max(\text{nextQValues})$$
$$Q_{\text{OtherValues}} = \{\text{q} \in \text{nextQValues} \mid q \neq Q_{\text{MaxNext}}\}$$

(3) **Compute the Sum of Other Q-values:**

$$\text{sumOthers} = \sum_{q \in Q_{\text{OtherValues}}} q$$

(4) **Calculate the Average Current Q-value of the Next State:**

$$Q_{\text{NextAvg}} = \frac{(1 - \text{explorationRate}) \cdot Q_{\text{MaxNext}} + \left(\frac{\text{explorationRate}}{|Q_{\text{OtherValues}}|}\right) \cdot \text{sumOthers}}{|\text{nextQValues}|}$$

where $|Q_{\text{OtherValues}}|$ is length of other values array, and $|\text{nextQValues}|$ is the length of all nextQValues array.

(5) **Define the Reward:**

$$\text{reward} = T_{\text{weight}} + S_{\text{weight}}$$

where:

- T_{weight}: Tactical alignment of the decision made for a given state.
- S_{weight}: Immediate success or failure of the action.

(6) **Update the Q-value for Non-terminal States:**

$$\text{Q(s,a)} = \text{Q(s,a)} + \alpha \cdot \left(\text{reward} + \gamma \cdot Q_{\text{NextAvg}} - \text{Q(s,a)}\right)$$

where $Q(s, a)$ is the current state action pair whose Q-value we are updating, α is the learning rate- currently 0.3- that determines how much the simulation learns from each action, and γ is the discount rate -currently 0.9- that determines how much the simulation takes the next state, hence, the potential cumulative success and failure into account in its calculation.

In the simulation's above equation, S_{weight} can be negative, representing unsuccessful actions taken by the agent such as a pass that is intercepted, a dribbling that gets tackled, or a shot that goes out of the goal range. Therefore *reward* can take negative values too in certain situations where T_{weight} + S_{weight} is negative. Then, the Q-value may also decrease, even become negative. The first reason why failure is taken into account in reward calculation is that the simulation utilizes Q-learning only for the team in possession when a decision needs to be made about what to do with the ball; hence, after a mistake and possession loss, the team who lost the ball is no longer in a valid state, and the team who tackled the ball is about to make a decision based on its own Q-table. This discontinuity encourages the punishment of the players for their mistakes because the other team's potential success after losing the ball does not directly affect the current team's Q-table. The second reason for punishing players for their mistakes is that it reflects the real-life evaluation of the players.

For terminal states, meaning that the current state's action ended up as a possession loss because the ball was intercepted, or it went out of bounds, or it got scored, we do not have a valid next state for the team that was in possession before the action. Instead, the next state will be the current state of the team who just gained possession. Therefore, for terminal states' Q-learning function, it is not needed to calculate the Q_{NextAvg}. We only need the following formula for the terminal states' Q-table updates:

$$\text{Q(s,a)} = \text{Q(s,a)} + \alpha \cdot (\text{reward} - \text{Q(s,a)})$$

As section 3.1.1 suggests, normalized and action-defined states helps to differentiate tactical weights. Table 3 below is an example differentiation for the current partitions of the states.

Table 3: Weights and Rewards for Dribbling Tactics

Weights/Tactics	Dribble More: "yes"	Dribble More: "no"
Tactical weight	Cat 1: 1, Cat 2: 1.5, Cat 3: 2, Cat 4: 2.5, Cat 5: 3	Cat 1,2,3,4,5: 0
Success reward	Cat 1: 2.7, Cat 2: 2.3, Cat 3: 2, Cat 4: 1.8, Cat 5: 1.2	Cat 1: 2.7, Cat 2: 2.3, Cat 3: 2, Cat 4: 1.8, Cat 5: 1.2
Failure reward	Cat 1: -1.2, Cat 2: -1.8, Cat 3: -2, Cat 4: -2.3, Cat 5: -2.7	Cat 1: -1.2, Cat 2: -1.8, Cat 3: -2, Cat 4: -2.3, Cat 5: -2.7

Categories (Cat 1,2,4,5) represent the state's dribbling rationality component. For instance, when the agent is in a state whose dribbling rationality category is 2 with "dribble more" tactic open, if he decides to dribble based on the Q-values and succeeds by arriving at the aimed location on the field without losing the ball, T_{weight} is 1.5, S_{weight} is 2.3; hence, $reward$ = 3.8 in the Q-learning formula. The idea is simple: if the user chooses "dribble more" as a tactic, assign a positive tactical reward to any dribbling decision but differentiate it between different rationality categories such that tactical reward is more when it is more rational to dribble. A different logic applies to the differentiation of success and failure impacts on reward. We reward more if the agent is less likely to succeed based on the rationality category because he achieved something unexpected, and we punish more if the agent is more likely to succeed based on the rationality category because he failed when he was more likely to succeed based on predetermined domain-based assumptions. Like this example, tactical weight and success weight differentiations are implemented for every state-action pair for both layers of Q-table, Table 1 and Table 2 (but with 451 states). Below are other examples:

Table 4: Weights and Rewards for Shooting Tactics

Weights/Tactics	Shoot More: "yes"	Shoot More: "no"
Tactical weight	Cat 1: 1, Cat 2: 1.5, Cat 3: 2, Cat 4: 2.5, Cat 5: 3	Cat 1,2,3,4,5: 0
Success reward	Cat 1: 12 or 2.7, Cat 2: 11 or 2.3, Cat 3: 10 or 2, Cat 4: 9 or 1.8, Cat 5: 8 or 1.2	Cat 1: 12 or 2.7, Cat 2: 11 or 2.3, Cat 3: 10 or 2, Cat 4: 9 or 1.8, Cat 5: 8 or 1.2
Failure reward	Cat 1: -1.2, Cat 2: -1.8, Cat 3: -2, Cat 4: -2.3, Cat 5: -2.7	Cat 1: -1.2, Cat 2: -1.8, Cat 3: -2, Cat 4: -2.3, Cat 5: -2.7

12,11,10,9,8 represent scoring a goal, whereas other numbers show on-target shots for success

For the second layer of the Q-table after a passing decision is made, Table 2, we need to decide how to pass based on the tactics. For Table 2, tactical weights are less dependent

Table 5: Weights and Rewards for Passing Based On Idea (Main Q-table)

Weights/Tactics	Idea: "Possession"	Idea: "Balanced" or "Counter"
Tactical weight	Cat 1: 1, Cat 2: 1.5, Cat 3: 2, Cat 4: 2.5, Cat 5: 3	Cat 1,2,3,4,5: 0
Success reward	Cat 1: 2.7, Cat 2: 2.3, Cat 3: 2, Cat 4: 1.8, Cat 5: 1.2	Cat 1: 2.7, Cat 2: 2.3, Cat 3: 2, Cat 4: 1.8, Cat 5: 1.2
Failure reward	Cat 1: -1.2, Cat 2: -1.8, Cat 3: -2, Cat 4: -2.3, Cat 5: -2.7	Cat 1: -1.2, Cat 2: -1.8, Cat 3: -2, Cat 4: -2.3, Cat 5: -2.7

Possession tactic would encourage passing more

on states' components but more dependent on the actions themselves such that "possession" would encourage to-feet and less direct passes, and "counter" would encourage to-space and direct passes. Table 6 displays how the tactical and success/failure weights are differentiated for Table 2.

Table 6: Weight Differentiation for the Second Layer Q-table with Respect to the Idea Tactic

weights/tactics	Idea=possession	Idea=counter	Idea=balanced
tactical weight	PTFL:7, PTSL:3, PTFD:0, PTSD:0	PTFL:0, PTSL:0, PTFD:3, PTSD:7	PTFL:0, PTSL:0, PTFD:0, PTSD:0
success reward	PTFL:4, PTSL:4, PTFD:6, PTSD:6	PTFL:4, PTSL:4, PTFD:6, PTSD:6	PTFL:4, PTSL:4, PTFD:6, PTSD:6
failure reward	PTFL:-6, PTSL:-6, PTFD:-4, PTSD:-4	PTFL:-6, PTSL:-6, PTFD:-4, PTSD:-4	PTFL:-6, PTSL:-6, PTFD:-4, PTSD:-4

It is important to note that these differentiations are for the current simulation, and further or different differentiations can be made to increase the realism and quality of the simulation. As long as the states are defined to have normalized components as in this simulation, it is convenient to organize states and tactical differentiations with respect to different levels of project scope.

3.1.3 Q-table Differentiation. We have seven tactics: creativity(low, balanced, high), mentality(very defensive, defensive, balanced, attack, all out attack), compactness (low, balanced, high), pressLevel (low, balanced, high, shootMore (yes,no), dribbleMore (yes,no), idea(counter, balanced, possession). As seen in the previous subsection, shootMore, dribbleMore, and idea directly affect the Q-learning formula through the reward variable because we award more or punish less based on these tactics as shown in Tables 3,4,5,6. For tactics that have a direct impact on the Q-learning function, it is observed that it is better to have a separate Q-table pair for each possible tactical combination for the following reasons.

- When there is only one Q-table pair with first and second layers for all tactical combinations, tactics have a minimal impact on game play. The simulation needs to be trained, and when a new tactic is implemented after an existing tactic that has been used for an important time, the new tactic would not affect decisions unless it has been used without changing it for an important time. Therefore, each tactical combination that has a direct impact on Q-learning formula should have its own learning tables, and when that combination is chosen by the user, the simulation should switch to the corresponding tables.
- It is not ideal to mix the learning of different tactical combinations into a single pair of Q-tables because the results and evaluations would be messy, not giving us the intended effect of tactics.

Therefore, because shootMore, dribbleMore, and idea can have 12 different combinations of values, we have 12 different pairs of Q-tables similar to Table 1 and Table 2, each having 451 states. The importance of this implementation is better understood in Sections 4 and 5, Training The Simulation and Evaluation, respectively.

3.2 Implementing Decided Actions

3.2.1 Shooting. Shooting is handled in three sequential parts: calculating the shooting target (to where the agent ends up shooting), initializing shooting, and updating shooting and its status per frame.

Calculating the shooting target, the x-value and y-value that the ball needs to reach in case of no interception by the opposition team- is based on the shot distance. x-value equals the goal's central x value because x represents the horizontal length of the pitch and a shot is always towards the goal's x value; however, y-value, where the ball will reach compared to the goal line, depends on the probabilities affected by the shot distance. As the shot distance approaches the maximum allowed distance for the players to shoot, 35 meters in our case, the probability of shooting towards a wider range of y-values [125,275] compared to the goal range [178,222] increases. However, the wider range is inclusive of the goal range, so even for the wider range, it is possible to find the goal as a target. This reflects real life because in real life it is easy to find the goal from close range and more difficult to find the goal from a far range, but it is still possible to find the goal in any shot. For both ranges, the exact y-value is chosen randomly. Here is how this logic is handled:

(1) **Define the Ranges:**

$$\text{Goal Range (y-coordinates): min: 178, max: 222}$$
$$\text{Extended Target Range (y-coordinates): min: 125, max: 275}$$

(2) **Calculate Distance from Goal:**

$$d = \sqrt{(x_{\text{shooter}} - x_{\text{goal}})^2 + (y_{\text{shooter}} - y_{\text{goal}})^2}$$

(3) **Calculate Favorability of Hitting the Goal:**

$$\text{favorGoal} = \max\left(0, 1 - \frac{d}{\text{allowedDistance}}\right)$$

where:

- d is the distance from the shooter to the goal.
- allowedDistance is the maximum distance allowed to shoot (default: 210-35 meters-).

(4) **Randomly Determine the Target y-Coordinate:**

$$r = \text{UniformRandom}(0, 1) \quad \text{(random factor)}$$

$$y_{\text{target}} = \begin{cases} \text{UniformRandom}(\text{Goal Range}_{\text{min}}, \text{Goal Range}_{\text{max}}), & \text{if } r < \text{favorGoal} \\ \text{UniformRandom}(\text{Target Range}_{\text{min}}, \text{Target Range}_{\text{max}}), & \text{otherwise.} \end{cases}$$

where:

- r is a random number between 0 and 1.
- If $r <$ favorGoal, the target y-coordinate is within the goal range.
- Otherwise, the target y-coordinate is within the extended target range.

(5) **Return the Target Position:**

$$(x_{\text{target}}, y_{\text{target}}) = (x_{\text{goal center}}, y_{\text{target}})$$

Initializing the shot phase is using the calculated target and information from the environment such as positions of the objects in the pitch to assign initial speed and acceleration values and to establish a unit distance for the ball's x and y values for the movement of the ball in the update phase. Then, all information needed to update shooting is passed to the updating step. Updating step checks the condition of the shot in every frame, if it is intercepted, scored, saved by the goalkeeper, reached out of bounds, or still in progress. If it is still in progress, it makes the ball move toward the target based on acceleration, deceleration, and current speed values. The ball starts to decelerate at half distance with a slower rate compared to the previous acceleration. If it is not in progress, it returns the corresponding status so that the simulation can handle the next steps.

3.2.2 Dribbling. Dribbling is handled in three sequential steps: calculating dribbling direction, initializing dribbling, and updating dribbling per frame.

Calculating the dribbling direction is based on two sub-weights: alignment with the attacking direction (the attacking direction or deviations from this direction) and misalignment from the direction towards the closest defender. For a unit circle of directions, we iterate through each degree of angle, by dividing the directions into 360 different directions, and for each direction, we check their alignment with the above-mentioned normalized sub-weights. Then, the final weight of each direction is the sum of these two sub-weights, but alignment with attack is given more importance because it is important to dribble towards the attacking direction in real life. Then, these weights are transformed into probabilities based on their sum, and dribble direction is chosen according to this probabilistic distribution: a higher probability to dribble to a direction with more weight and a lower probability to dribble towards a direction with less weight. Here are the details:

(1) **Generate Possible Directions:**

$$\theta_i = \frac{i \cdot 2\pi}{\text{numSamples}} \quad \text{for } i = 0, 1, \ldots, (\text{numSamples} - 1)$$

where θ_i represents the direction in radians and numSamples equals 360.

(2) **Calculate Weights for Each Direction:**

(a) **Alignment with Attacking Direction:**

$$\mathbf{dirVector} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$$

$$\mathbf{attackingVector} = \begin{bmatrix} \text{attackingDirection} \\ 0 \end{bmatrix}$$

$$\text{dotAttacking} = \mathbf{dirVector} \cdot \mathbf{attackingVector} = \cos(\theta) \cdot \text{attackingDirection}$$

$$\text{alignmentWithAttack} = \max(0, \text{dotAttacking})$$

(b) **Avoidance of the Closest Defender:**

$$\mathbf{defenderVector} = \begin{bmatrix} \text{closestDefender.x} - \text{player.x} \\ \text{closestDefender.y} - \text{player.y} \end{bmatrix}$$

$$\|\mathbf{defenderVector}\| = \sqrt{(\text{defenderVector.x})^2 + (\text{defenderVector.y})^2}$$

$$\mathbf{normalizedDefender} = \frac{\mathbf{defenderVector}}{\|\mathbf{defenderVector}\|}$$

$$\text{dotAwayFromDefender} = -(\mathbf{dirVector} \cdot \mathbf{normalizedDefender})$$

$$\text{alignmentWithDefender} = \max(0, \text{dotAwayFromDefender})$$

(c) **Combine Weights:**

$$\text{weight} = 0.6 \cdot \text{alignmentWithAttack} + 0.4 \cdot \text{alignmentWithDefender}$$

(3) **Normalize Weights:**

$$\text{totalWeight} = \sum_i \text{weight}_i$$

$$\text{probability}_i = \frac{\text{weight}_i}{\text{totalWeight}}$$

(4) **Sample a Direction Based on Probabilities:**

$$\text{cumulativeProbability}_i = \sum_{j=1}^{i} \text{probability}_j$$

$$\text{randomValue} = \text{UniformRandom}(0, 1)$$

$$\text{chosenIndex} = \min\{i : \text{randomValue} \leq \text{cumulativeProbability}_i\}$$

$$\text{chosenDirection} = \theta_{\text{chosenIndex}}$$

(5) **Return the Chosen Direction as a Vector:**

$$\mathbf{chosenVector} = \begin{bmatrix} \cos(\text{chosenDirection}) \\ \sin(\text{chosenDirection}) \end{bmatrix}$$

The initializing dribbling step uses the calculated dribbling direction and the environmental factors in the simulation to assign a randomized dribbling distance to the dribbling direction and then passes all dribbling and environmental information necessary to update dribbling to the updating phase. Updating the dribbling phase checks the status of the dribbling per frame, if it is tackled, if it ended up as a goal scored (edge case), if the ball is out of bounds of the field, if the dribbling ended up as successful (the player reached to the intended distance), or if it is still in progress. For in-progress dribbling, it updates the position of the ball and the dribbler. For other statuses, it returns the appropriate status for the simulation to know the next steps.

3.2.3 Passing. Passing action has its own sub-actions as shown in Table 2: "PTFL" (passing to feet and passing to a less direct target with respect to the attacking direction), "PTSL" (passing to space and passing to a less direct target with respect to the attacking direction), "PTFD" (passing to feet and a more direct pass), "PTSD" (passing to space and a more direct pass). First, a target player is determined based on the directness of the pass (less direct or direct). Second, a target position to pass is calculated based on the type of pass (to-feet or to-space) with respect to the target player. Third, the pass is initialized. Fourth, the pass is updated for each frame.

Determining the target player for less direct passes is based on finding the most available four teammates with respect to their distances from the closest opposition player to them and with respect to how clear the line from the passer to the teammate is (checking any opposition player is in between), and then assigning weights to these four teammates based on proximity to the passing player to provide a probabilistic distribution in which we give more probability to pass towards the closer ones. Here is the algorithm:

(1) **Find the 4 Most Available Teammates:**
  (a) **Calculate Availability Scores:**

$$\text{availabilityScore(teammate)} = \begin{cases} \text{distanceFromDefender} & \text{if line is clear} \\ \frac{\text{distanceFromDefender}}{2} & \text{otherwise} \end{cases}$$

$$\text{distanceFromDefender} = \|\text{teammate} - \text{closestDefender}\|$$

  (b) **Sort by Availability:**

sortedAvailability = Sort descending by availabilityScore and take the top 4.

(2) **Assign Weights Based on Proximity to Passing Player:**
  (a) **Total Proximity:**

$$\text{totalProximity} = \sum_{\text{teammate} \in \text{sortedAvailability}} \|\text{passingPlayer} - \text{teammate}\|$$

  (b) **Weight Assignment:**

$$\text{weight(teammate)} = \begin{cases} 1 - \frac{\|\text{passingPlayer} - \text{teammate}\|}{\text{totalProximity}}, & \text{if totalProximity} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

(3) **Normalize Weights to Probabilities:**

$$\text{weightSum} = \sum_{\text{teammate} \in \text{sortedAvailability}} \text{weight(teammate)}$$

$$\text{normalizedWeight(teammate)} = \begin{cases} \frac{\text{weight(teammate)}}{\text{weightSum}}, & \text{if weightSum} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

(4) **Randomly Select a Target Player Based on Probabilities:**
  (a) **Cumulative Probabilities:**

$$\text{cumulativeProb}_i = \sum_{j=1}^{i} \text{normalizedWeight}(\text{teammate}_j)$$

  (b) **Random Selection:**

$$\text{randomValue} \sim \text{Uniform}(0, 1)$$

$$\text{chosenIndex} = \min\{i \mid \text{randomValue} \leq \text{cumulativeProb}_i\}$$

  (c) **Return the Chosen Teammate:**

$$\text{chosenTeammate} = \text{teammate}_{\text{chosenIndex}}$$

Determining the target player for direct passes is based on finding the four teammates that are closest to the goal being attacked, then assigning weights to these four teammates with respect to their availability (distance from the closest defender and passing line clarity from the opposition) to use these weights as probabilities in determining the target player, giving more probability to pass to a more available teammate among the four. Here is the algorithm:

(1) **Find the 4 Closest Teammates to the Attacking Goal:**

$$\text{goalDistances} = \{\text{teammate}, \text{distance} = \|\text{teammate} - \text{attackingGoal}\| \mid \text{teammate} \in \text{teammates}\}$$

$$\text{sortedByGoalDistance} = \text{Sort(goalDistances by distance ascending)}[:4]$$

(2) **Assign Weights Based on Availability:**

$$\text{weight(teammate)} = \begin{cases} \text{defenderDistance}, & \text{if clear path} \\ \text{defenderDistance} \cdot 0.5, & \text{otherwise} \end{cases}$$

$$\text{defenderDistance} = \|\text{teammate} - \text{closestDefender}\|$$

(3) **Adjust Weights to Favor "Clear Path" Players in Any Case:**

$$\text{maxBlockedWeight} = \max\left(\text{weight(teammate)} \mid \text{teammate.clear} = \text{false}\right)$$

$$\text{adjustedWeight(teammate)} = \begin{cases} \text{weight(teammate)} + \text{maxBlockedWeight} + 1, & \text{if clear path} \\ \text{weight(teammate)}, & \text{otherwise} \end{cases}$$

(4) **Normalize Weights to Probabilities:**

$$\text{totalWeight} = \sum_{\text{teammate} \in \text{adjustedWeight}} \text{adjustedWeight(teammate)}$$

$$\text{normalizedWeight(teammate)} = \begin{cases} \frac{\text{adjustedWeight(teammate)}}{\text{totalWeight}}, & \text{if totalWeight} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

(5) **Randomly Select a Target Player Based on Probabilities:**
  (a) **Cumulative Probabilities:**

$$\text{cumulativeProb}_i = \sum_{j=1}^{i} \text{normalizedWeight}(\text{teammate}_j)$$

  (b) **Random Selection:**

$$\text{randomValue} \sim \text{Uniform}(0, 1)$$

$$\text{chosenIndex} = \min\{i \mid \text{randomValue} \leq \text{cumulativeProb}_i\}$$

  (c) **Return the Chosen Teammate:**

$$\text{chosenTeammate} = \text{teammate}_{\text{chosenIndex}}$$

After the target player is determined by using one of the determining algorithms based on the directness of the pass, a target position is calculated by using the target player. For to-feet passes, we first find the distance between the passing player and the target player, and then assign a passing radius centering around the target player based on this distance such that the radius gets larger as the distance increases so that the longer passes have higher chances to shift from the target player's position, reflecting the real life. Then, within this radius, a random point is selected as the target position. Here is the complete implementation:

(1) **Calculate the Distance Between Passing Player and Target Player:**

$$d = \sqrt{(\text{passingPlayer}_x - \text{targetPlayer}_x)^2 + (\text{passingPlayer}_y - \text{targetPlayer}_y)^2}$$

(2) **Determine the Radius for the Pass:**

$$r = \min\left(\frac{d}{10}, \text{maxRadius}\right)$$

where maxRadius is the maximum allowable radius (default: 18 -3 meters in real life).

(3) **Get the Random Angle and Distance within Radius:**

$$\theta = \text{Random}(0, 2\pi) \quad \text{(Random angle in radians)}$$
$$r_{\text{random}} = \text{Random}(0, r) \quad \text{(Random distance within the radius)}$$

(4) **Calculate the Target Position According to the Random Point in Radius:**

$$\text{targetPosition}_x = \text{targetPlayer}_x + r_{\text{random}} \cdot \cos(\theta)$$
$$\text{targetPosition}_y = \text{targetPlayer}_y + r_{\text{random}} \cdot \sin(\theta)$$

For to-space passes, on the other hand, the logic is a little different. The radius centering around the target player is limited to the semicircle that faces the goal being attacked. This is done to ensure that to-space passes are towards the goal side to reflect real-life passes played to the running path of the players during attacking. Besides, the radius is also doubled compared to the to-feet passes to implement to-space logic. It is still possible for the pass to act like a to-feet pass toward the attacking goal because to-space radius is inclusive of to-feet radius; however, this also reflects real-life passing because a player cannot always provide a perfect to-space pass. Here is the complete implementation:

(1) **Calculate the Distance Between the Passing Player and the Target Player:**

$$d = \sqrt{(\text{passingPlayer}_x - \text{targetPlayer}_x)^2 + (\text{passingPlayer}_y - \text{targetPlayer}_y)^2}$$

(2) **Determine the Radius for the Pass:**

$$r = \min\left(\frac{d}{10}, \text{maxRadius}\right) \cdot 2$$

where maxRadius is the maximum allowable pass radius (default: 18).

(3) **Define the Attacking Direction:**

$$\text{G}_{\text{direction}} = \begin{bmatrix} \text{goal.xCenter} - \text{targetPlayer.x} \\ \text{goal.yCenter} - \text{targetPlayer.y} \end{bmatrix}$$

(4) **Normalize the Goal Direction Vector:**

$$\|\text{G}_{\text{direction}}\| = \sqrt{(\text{G}_{\text{direction},x})^2 + (\text{G}_{\text{direction},y})^2}$$
$$\text{G}_{\text{normalized}} = \begin{bmatrix} \frac{\text{G}_{\text{direction},x}}{\|\text{G}_{\text{direction}}\|} \\ \frac{\text{G}_{\text{direction},y}}{\|\text{G}_{\text{direction}}\|} \end{bmatrix}$$

(5) **Randomly Select a Point in the Semicircle Towards the Attacking Goal:**

$$\text{angleOffset} = \text{Random}(-\pi/2, \pi/2)$$
$$\theta = \arctan\left(\frac{\text{G}_{\text{normalized},y}}{\text{G}_{\text{normalized},x}}\right) + \text{angleOffset}$$
$$r_{\text{random}} = \text{Random}(0, r)$$

(6) **Calculate the Target Position:**

$$\text{targetPosition}_x = \text{targetPlayer}_x + r_{\text{random}} \cdot \cos(\theta)$$
$$\text{targetPosition}_y = \text{targetPlayer}_y + r_{\text{random}} \cdot \sin(\theta)$$

Initializing the pass phase uses the calculated target position based on the above algorithms and environmental factors to establish randomized acceleration and deceleration rates and to indicate unit x and unit y distances for the ball to travel in the update phase. Then, all information necessary to update passing is passed to the update phase. The update phase checks the status of the pass, if it is intercepted, if it ended up as a goal scored (edge case), if it is out of field bounds, if it is completed (reached the target position), or if it is still in progress. It returns the appropriate status if it is not in progress so that the simulation can handle the next steps, and if it is still in progress, it updates the ball's position.

3.3 Movements of the Other Players

3.3.1 In Possession Movement. Movements of the players in the team that is currently attacking are handled by a rule-based AI. The target player of the pass moves towards the ball with a normal player speed, adjusted as 0.5 per frame, corresponding to traveling 5 meters per second in real life considering the size adjustments of the field in the simulation, a realistic and enjoyable speed for the user. We know that a new decision should be made when the target player gets the ball, so we return to our Q-tables and decision-making policies. Other than the target player, all other players' movements are affected by the following factors: move-up factor, move-down factor, unmarking factor, and original position factor. These factors change between positions, and for certain cases, how they are utilized is also different. The Move-up factor is used to move towards the attacking direction as the ball moves forward in the pitch; the move-down factor is used to move towards the defending direction as the ball moves towards the defending direction on the pitch. The unmarking factor is a factor to move away from the closest defender. And original position factor is a factor ensuring that players do not lose their positions while trying to adjust their movements based on the other factors. These factors are affected by the tactics given by the user. For instance, mentality ("park the bus", "defense", "balanced", "attack", "all out attack") affects the move-up factor; the more attacking, the higher factor. The Move-down factor decreases or increases according to the move-up factor; both factors sum to the same number all the time. Creativity tactic ("low", "balanced", high") affects both the unmarking factor and the original position factor. The higher the creativity, the larger the unmarking factor and the smaller the original position factor. All players consider off-side, and they are programmed not to be off-side for now. All four factors are values between 0 and 1, and they work by increasing/decreasing the x and y values of the players based on a unit distance towards the position the factor is representing, multiplied by the factor itself, multiplied by the normal player speed, multiplied by the distance effect. Here is an example:

$$midfielder.x+ = unitXForOriginal*distanceEffectOriginal$$
$$*originalPositionFactor * normalPlayerSpeed$$
$$midfielder.y+ = unitYForOriginal*distanceEffectOriginal$$
$$*originalPositionFactor * normalPlayerSpeed$$

Normal player speed is 0.5, and all other variables are less than or equal to 1, so the player speed is less than or equal to normal player speed. Unit x and unit y only represent a unit distance between 0 and 1 per frame to the target's x and y to adjust moving per frame, and the distance effect is calculated as below:

$$\text{distanceEffect} = \begin{cases} \frac{\text{distance}}{\text{maxNormalize}}, & \text{if distance} \leq \text{maxNormalize (67 meters)}, \\ 1, & \text{if distance} > \text{maxNormalize(67 meters)}. \end{cases}$$

In the above example of the midfielder's movement affected by the original position, the distance would be the distance from the original position, and the distance effect would be a normalized effect that increases as the distance from the original position increases. Distance effect is used to boost the factors- move up factor, move down factor, original position factor, unmark factor- against each other because if the distance to the point one of the factors is representing is a lot, the distance effect for that specific factor will be higher than the other factors, increasing its impact and balancing the overall movement. Just like the above example, the other three factors' impacts on the movement are also calculated with the same logic. The four factors are not originally same and they can differ from tactic to tactic too, and a dominant factor is balanced by the corresponding distance effects of each factor per frame to adopt the conditions in the field.

Defenders are affected more by their original positions so that they do not leave defense much, and their unmarking factor is also less than those of the midfielders and attackers.

3.3.2 Out Of Possession Movement. The closest player to the ball presses the ball to tackle or intercept it. The press level tactic ("low", "balanced", "high") affects the presser's speed. If the ball is closer to the goalkeeper than a certain distance, the goalkeeper's y value, his position on the goal line, changes towards the ball's y value so that the keeper can save the ball coming towards the goal. Otherwise, the goalkeeper moves towards his original position. The movements of all players other than the pressing player and the goalkeeper, the rest of the defenders, midfielders, and attackers, depend on the following logic and factors.

- Each player moves towards the average team position with respect to the pressing factor and compactness factor. The pressing factor affects the player's x value while the compactness factor affects the player's y value when moving towards the average team position. Compactness, therefore, is more about staying close with respect to the height (height represents the shorter edge of the rectangle field in our case), and the pressing factor is more about getting closer to each other with respect to the length of the pitch. The average team position (x,y) means the average position of all players in the team. Here, it is important to notice that while the presser is pressing the ball, it changes the average position of the team to some extent; therefore, the logic behind moving towards the average position is not just to stay together as a team but also to move towards the ball's location.
- Each player moves towards their original position with respect to the original position factor.
- Each player marks an opposition player that is currently not marked and moves towards him with respect to the press factor.

As the in-possession movement, all factors, press factor, compactness factor, and original position factor, -together with unit x and unit y distances to the points the factors are representing and the normalized distance effects of each factor- multiply the normal player speed to move the players. The defenders' and attackers' original position factors are higher than their press factors and compactness factors; while midfielders' factors are more balanced. Tactics also affect these factors. Defensive Compactness tactic given by the user ("low", "balanced", "high") affects the compactness, press level ("low", "balanced", "high") affects the press factor, and creativity ("low", "balanced", "high") affects the original position factor such that the higher the creativity, the smaller the original position factor. Besides, when defenders are in the attacking half, and the ball is lost, the defenders' original position factor, differently from the midfielders and the attackers, increases to 1 to dominate the other factors so that they can run back immediately. Additionally, when the defenders are in their own half on the field but the ball is closer to the goal they are defending than themselves, the defenders' original position factor, differently from the midfielders and attackers, decreases to 0 so that they can move towards the ball without caring about their original position.

3.4 Dead-ball Handling

Goal-kicks, kick-offs, throw-ins, free-kicks, and penalties are handled separately. They are not part of the Q-tables. Their conditions are checked per frame, and they are implemented only when their condition is true. They initiate a pass or shoot, and then, the simulation comes back to making decisions based on the Q-tables and handling the movements of the other players. For the pass coming from a goal-kick, kick-off, throw-in, and free-kicks and for the shots coming from free-kicks and penalties, the Q-tables are not updated as the current state that needs to be updated would be invalid in the Q-table, because they are handled separately.

For fouls ending up as a free-kick or penalty, the following methodology is implemented.

- The vector from the attacker to defender is defined
- The dribble direction vector is computed based on the dribbling implementation as seen in Section 3.2.
- The angle between these two factors is calculated.
- A weight that is used in foul possibility per angle is calculated such that the weight is higher if the angle equals π and lower if as the angle deviates from π because it is more possible for a player to commit a foul if he directly presses from behind of the attacker when trying to get the ball.

Here is the algorithm used:

(1) **Compute Relative Position and Distance:**

$$\text{relX} = \text{defender.x} - \text{attacker.x},$$
$$\text{relY} = \text{defender.y} - \text{attacker.y},$$
$$\text{dist} = \sqrt{\text{relX}^2 + \text{relY}^2}.$$

(2) **Handle Perfect Overlap:**

$$\text{If dist} = 0, \quad \text{then return } 0.$$

(3) **Compute Cosine of Angle:**

$$\cos\theta = \frac{\text{dribbleDirection.x relX} + \text{dribbleDirection.y relY}}{\text{dist}},$$
$$\cos\theta \leftarrow \min(1, \max(-1, \cos\theta)).$$

(4) **Recover Angle θ:**

$$\theta = \arccos(\cos\theta).$$

(5) **Compute Foul Weight:**

$$\text{foulWeight} = \frac{\theta}{\pi}.$$

3.5 Graphical Implementation

Node.JS is used to create the animations for the simulation. In the future, this simulation is planned to be enhanced further and re-developed by using Unity for a 3D environment.

4 TRAINING THE SIMULATION

The project is enhanced with simulation training, meaning that Q-values are not initialized as 0 when a simulation starts, but initialized with trained values to enhance realism. In fact, for now, the simulation is designed such that there is a constant training every time the simulation is working. Our 12 different pairs of Q-tables per tactical combination is stored in a file with respective keys, and when a user changes a tactic mid-game, the simulation loads that tactic's Q-table from the file and saves it periodically to update the trained values. For instance, if the user switches to the tactic shootMore:yes, dribbleMore:No, idea:Possession, the simulation loads the respective Q-tables for this specific tactical combination. Then, periodically, the simulation saves the Q-tables to the file. I/O operations are carefully serialized, temporary files are utilized, and last error-free Q-table is maintained all the time to not to get affected from possible load/save errors that would change the training data and cause losses.

Ensuring that each tactical combination has its own learning, and this learning is constantly being trained without a loss whenever the simulation is on makes our simulation better and more realistic each day.

Especially when we apply training, we understand the necessity of having different learning tables per a tactical combination. If the simulation is only trained for balanced tactics for days, and then the user switches tactics in the game, those tactics would not even affect the players. Or if the simulation is trained with varying tactics but with a single pair of Q-tables, the training would not mean anything because we would not be able to evaluate how and to what extent each tactic affects the decisions made.

5 EVALUATION

.

(1) Qualitatively, user testing sessions were conducted, and verbal feedback was gathered. Initially, the test subjects found the game to be too fast, especially the passing speed. The re-calculations were made to check if the passing acceleration and deceleration reflected real life, and it was realized that they were indeed chosen to be slightly more than they should have. Then, the passing speed was adjusted accordingly.

(2) Quantitatively, training the simulation helped evaluations we made by automating the process. The further the simulation was trained, the more realistic the game became, as also observed from the Q-values. Before training the simulation, when there was only in-game learning, it was seen that the values were changing too fast, and they were fluctuating more than they should have. Then, the tactical weights and success/failure weights were re-adjusted to solve the issue, but this solved the issue only to some extent. Then, training the simulation and observing and improving Q-tables periodically showed that the players stopped doing unnecessary mistakes because the Q-values per state-action pair increased. In fact, before training, these values were negative, which was a major problem during Capstone Project 1, the initial version of this project. Right now, many of these values are positive, suggesting that players play more realistically, avoiding some unrealistic decisions that lead to mistakes. This was also verified by feedback gathered from the user testing sessions conducted periodically.

(3) Luckily, the main claim of this project is the modifiability of the states. By changing the partition of the action-defined component categories within each state, the simulation can easily multiply its number of states and create much more differentiation. Thanks to this differentiation, we can also further differentiate the

tactical weights per state-action pairs, enhancing both the realism and the quality of the project. In fact, the initial prototype had 37 states as shown in Table 1 and Table 2. However, currently, the project has 451 states, and this also affected Q-values positively, fixing the problem of too many negative Q-values. Therefore, the main claim of this project, its modifiability is self-evident in its methodology, and the project can even be further improved.

(4) The effect of the tactics on the simulation is also evaluated with algorithms comparing the Q-tables of 12 different possible tactical combinations. Currently, the simulation is trained for 1000 minutes, and the below analysis and tables display that the simulation is successfully harmonizing tactics and decision-making.

5.1 Quantitative Evaluation

First, we check if the tactics are affecting Q-values as intended from a broad perspective. Second, we check if tactics actually encourage the actions that they are supposed to encourage in Q-learning formula by analyzing each tactic's Q-tables separately and checking which action has the highest Q-values overall within 451 states, and then comparing each tactic.

5.1.1 Checking The Q-learning Formula and Tactic's Impact on Q-values From a Broader Perspective. Recall that the reward variable within Q-learning formula is affected both by the tactical alignment that is differentiated among states and also success/failure. This suggests that, if we have a tactic that has shootMore: Yes, its Q-values for shooting will be systematically higher compared to a Q-table of a tactic that has shootMore: No because when a shot fails, we punish it through our fail weight, but we also award with tactical alignment weight in case of shootMore: Yes, decreasing the overall punishment and resulting in a higher Q-value. Therefore, we first check that if our tactics and Q-learning formula works as intended by comparing each tactic's Q-table's first layer values per state and checking which tactic had the highest Q-values overall per state and assigning a win rate, which is what percentage of states that specific tactic had the highest Q-values. Table 7 below displays this win rate, and not surprisingly, shootMore:yes, dribbleMore: yes, and idea:possession each increases the win rate when the tactical combinations are combined because each of them are encouraging the corresponding action taken within the Q-learning formula by decreasing the punishments of the mistakes. Therefore, it is no surprise that "yes_yes_possession" has 0.88 winning rate; in fact, it is what we want to see because it shows that our tactical differentiation and Q-learning formula work as expected.

Table 7: Overall Win-Rates per Tactic (First Layer)

Tactic	Win-Rate
yes_yes_possession	0.8777
yes_yes_balanced	0.0110
yes_yes_counter	0.0086
yes_no_possession	0.0196
yes_no_balanced	0.0110
yes_no_counter	0.0102
no_yes_possession	0.0243
no_yes_balanced	0.0110
no_yes_counter	0.0071
no_no_possession	0.0141
no_no_balanced	0.0039
no_no_counter	0.0016

5.1.2 Checking the Tactics' Encouragement of Actions. For each tactic t we load its first-layer Q-table

$$Q_t^1 : \mathcal{S} \times \mathcal{A}^1 \to \mathbb{R}$$

and second-layer Q-table

$$Q_t^2 : \mathcal{S} \times \mathcal{A}^2 \to \mathbb{R},$$

where $\mathcal{S}$ is the (shared) set of 451 discrete states, $\mathcal{A}^1 = \{\mathit{shoot}, \mathit{dribble}, \mathit{pass}\}$, and $\mathcal{A}^2 = \{\mathit{passToFeetLessDirect}, \ldots, \mathit{passToSpaceDirect}\}$.

Procedure.

(1) For each tactic t, accumulate the sum of all Q-values for each action over all states, then divide by the number of states to obtain the *average* Q-value of that action.

(2) Identify the action with the *highest* average Q-value for each tactic, in both layers.

The results are shown in Tables 8 and 9.

Table 8: First-Layer: average Q-values per action, per tactic. Best action in bold.

Tactic	Pass	Dribble	Shoot
yes_yes_possession	**0.293**	0.125	0.071
yes_yes_balanced	0.047	0.098	**0.113**
yes_yes_counter	0.051	**0.094**	0.091
yes_no_possession	**0.301**	-0.022	0.120
yes_no_balanced	0.064	-0.040	**0.152**
yes_no_counter	0.066	-0.028	**0.129**
no_yes_possession	**0.293**	0.120	0.002
no_yes_balanced	0.064	**0.105**	-0.023
no_yes_counter	0.045	**0.099**	-0.003
no_no_possession	**0.297**	-0.037	0.001
no_no_balanced	**0.070**	-0.056	-0.003
no_no_counter	**0.041**	-0.054	-0.003

Table 9: Second-Layer: average Q-values per pass-type action, per tactic. Best action in bold.

Tactic	Feet–Less (Indirect)	Feet–Direct	Space–Less (Indirect)	Space–Direct
yes_yes_possession	**0.570**	0.108	0.181	0.103
yes_yes_balanced	0.065	**0.094**	0.036	0.052
yes_yes_counter	0.072	0.238	0.056	**0.397**
yes_no_possession	**0.572**	0.153	0.174	0.121
yes_no_balanced	0.074	0.090	0.027	**0.101**
yes_no_counter	0.094	0.302	0.056	**0.506**
no_yes_possession	**0.496**	0.147	0.155	0.078
no_yes_balanced	0.055	0.050	0.018	**0.105**
no_yes_counter	0.062	0.215	0.046	**0.402**
no_no_possession	**0.586**	0.123	0.198	0.093
no_no_balanced	0.016	0.097	0.062	**0.121**
no_no_counter	0.038	0.267	0.046	**0.458**

Table 8 suggests that shootMore: "yes" successfully favors shoot, dribbleMore: "yes" successfully favors dribble, and idea: "possession" successfully favors pass within the Q-table of each tactic. Here, while comparing different tactics, rather than comparing the values directly, we should compare which actions were more favored per tactic, and if that align with the intended logic. The tactics "no_yes_balanced" and "no_yes_counter" show that dribbleMore: "yes" successfully favors dribble. The tactics "yes_no_balanced" and "yes_no_counter" show that shootMore: "yes" successfully favors shoot. The fact that idea: "possession" favors pass even when one or both of the other tactics are "yes" shows that idea: "possession" successfully favors pass. In general, we see that passing is favored more where more balanced tactics are chosen, and this also reflects real life football because it is usually a safer option.

Table 9 suggests that idea: "possession" successfully favors less direct and to-feet passes while idea: "counter" successfully favors direct and to-space passes.

These tables suggest that our training methodology, Q-table differentiations, state differentiations, and tactical differentiations work as we want and expect. Besides, notice that, regardless of which value is favored, there is not a very big difference between Q-values within each tactic's own Q-tables. This is also wanted and expected because we want our tactics to impact our decisions, but not entirely. Success/failure of the decisions also impact Q-values and success/failure and tactical alignment work in harmony, also in harmony with the exploration rate which gives a will for our agents to explore different actions and learn from them.

6 FUTURE WORK

- A possible approach is to develop individualized tactics per player on the team to affect movements and Q-table decisions, further decentralizing the learning and decision-making and enhancing realism.
- The simulation can be transformed to work in a 3D environment, such as Unity

7 PROJECT TIMELINE

What Were Done in Semester 1:

- Month 1: Literature review and finalization of project scope.
- Month 2-4: Development of Q-learning Strategy and algorithms as well as more basic rule-based AI algorithm
- Month 3-4: Integrating all different modules and creating basic graphical implementation of the simulation
- Month 4: High level evaluations to enhance the simulation

What were done Semester 2:

- Month 1-2: Implementing fouls, increasing state and tactical differentiation, and improving movements
- Month 3-4: Differentiating 12 Q-table pairs per tactical combination, training the simulation per tactical combination, developing quantitative evaluation summarized in Tables 7, 8, and 9, ensuring that all Semester 2 updates are harmonized without problems

8 CONCLUSION

This project aims to provide insights into the current football gaming industry and potential future real-life football simulations through an efficient Q-learning-based approach. The simulation focuses on designing Q-tables that harmonize tactical weights and success/failure-based rewards while offering a modifiable and compatible method for expanding states and incorporating new ideas. The user's tactical inputs influence the team's behavior by affecting the evaluation of agents' actions, with players having the agency to explore and learn from their decisions. The reward system considers immediate success/failure and potential future outcomes based on Q-values. The project introduces a unique state definition based on predetermined action rationality scores, which inform state representation without dictating agent decisions. A two-layer Q-table system is implemented for passing decisions, and player movements without ball possession are governed by rule-based AI algorithms. The simulation aims to reflect real-life soccer dynamics with randomization for enhanced smoothness, providing insights for future enhancements in football gaming and analysis.

REFERENCES

[1] Gürdal Arslan and Serdar Yüksel. 2017. Decentralized Q-Learning for Stochastic Teams and Games. *IEEE Trans. Automat. Control* 62, 4, 1545–1558. https://doi.org/10.1109/TAC.2016.2598476

[2] Richard Bellman. 1966. Dynamic programming. *science* 153, 3731, 34–37.

[3] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. 2019. Q-Learning Algorithms: A Comprehensive Classification and Applications. *IEEE Access* 7, 133653–133667. https://doi.org/10.1109/ACCESS.2019.2941229

[4] David S. Leslie and E. J. Collins. 2005. Individual Q-Learning in Normal Form Games. *SIAM Journal on Control and Optimization* 44, 2, 495–514. https://doi.org/10.1137/S0363012903437976 arXiv:https://doi.org/10.1137/S0363012903437976

[5] Purvag G. Patel, Norman Carver, and Shahram Rahimi. 2011. Tuning computer gaming agents using Q-learning. In *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 581–588.

[6] Robert Rein and Daniel Memmert. 2016. Big data and tactical analysis in elite soccer: future challenges and opportunities for sports science. *SpringerPlus* 5. https://doi.org/10.1186/s40064-016-3108-2