FINAL PROJECT
CS 687- FALL 2024
PARTH GOEL

Project Description

For this project I have decided to implement a **new MDP**.
I have implemented 2 existing RL methods that we learned in class on it in order to solve it and then gauge its performance.

- Q-learning
- Sarsa

Along with this, I have implemented an algorithm that was not covered in class:

- True-Online Sarsa

The purpose of it is to compare how well it performs to the methods we have covered in class and learn the difference between them.

MDP

Summary

The problem that I have chosen to model as an MDP is a Football Simulator. The idea is that given an environment of a field and number of opposition defenders placed in certain positions, a player should be able to over time learn the best decision to make in order to score a goal. This decision can be whether in the form of path-planning or by taking into consideration the probability of a shot from a certain location to result in a goal.
This can be useful in real-world applications as sports teams place heavy value on analytics for strategy and tactics. Learning the best decisions to make for any scenario beforehand would give a team an edge over their opponents.

Description
The MDP is defined as such:

Environment:
Consists of a NxM field with an opposition goal located at (N, M/2). (0,0) refers to the Top-left, (N,M) refers to the Bottom-Right. There are k opposition defenders that are positioned on the field each with their own (x,y) coordinate. There cannot be duplicates i.e. 2 defenders in the same position. In order to simulate real-world conditions and increase the difficulty of the player, the defenders are all positioned in their own half i.e. x >= N/2.
The Ball is in possession of the player ie. agent and hence is not explicitly defined.

States:
Each (x,y) coordinate on the grid defines a possible position or 'state' of our player. More specifically, x E [0, N] and y E [0, M].

Initial State: The player is initialized randomly and similar to the defenders, for realism is positioned in his own half i.e. x <= N/2. The player cannot be initialized in the same position as a defender.

Terminal State: The terminal state is implicitly defined as whenever the player loses the ball i.e. through a defender or via shooting. The episode is restarted upon reaching a terminal state.

Actions:
The player can take 5 possible actions at any time step:
Up, Down, Left, Right and Shoot.
The probability of any action succeeding in taking place is always 1.

Dynamics:
This version of the MDP is deterministic. There are several feature functions that are needed in order to effectively simulate a football match situation.

- If a ‘move’ action is taken i.e. up, down, left or right; the player moves in the specified direction unless the action takes it off the grid in which case, it remains where it was.

- Upon moving, if a player is deemed to be too close to a defender by a certain distance threshold ‘delta’, the ball is lost and the player instead transitions to the terminal state.
  The formula used here is: euclidean_distance(player, defender) < delta, for all defenders.

- If a ‘shoot’ action is taken the player transitions to the terminal state. In order to gauge whether a shot results in a goal, the probability of success depends on the distance of the player from the goal.
  The probability is defined as: max(0.1, 1 - (euclidean_distance(player, goal) / M) . *Division by M is done for normalization. The further away the player is, the lower the probability of scoring.

Along with this, in order to simulate realism, an extra feature is added that a shot will be blocked if there exists a defender in the line, or close enough by a certain distance threshold delta to the line between the player and the goal. This is done via the following code:

```
        px, py = player
        gx, gy = goal
        ox, oy = opponent
    if min(px, gx) < ox < max(px, gx):
        y_loc = ((ox - px)*((gy - py)/(gx - px))) + py
        if (y_loc > oy and oy + delta > y_loc) or (y_loc < oy and oy - delta < y_loc):
            print(f"Shot Blocked! Shot from: {player}, and blocked by: {opponent}")
            return True
    return False
```

<u>Rewards:</u>
The reward is always -0.05. This is done instead of 0 to reward the player for finding a way to score faster rather than dallying on the ball.
Successful shot resulting in goal: 10
Missed shot : -5
Blocked Shot: -8
Ball lost via getting close to opponent: -10

This reward structure incentivizes the player to not lose the ball by running into defenders, not take shots that have no chance of going in due to a defender being in the way and to also not take lucky shots from afar.

<u>Global Parameters:</u>
- Field size (NxM)
- No. of Defenders (k)
- Delta i.e. Defending radius.

The aim is for the player to learn based on his position and the position of the players, what is the position to shoot from and best path to take to that shooting position such that the shot has the highest chance of resulting in a goal.

RL Algorithms

SARSA

I implement the sarsa algorithm that we learned in class on this MDP. The algorithm updates the Q-values of the MDP based on a single step using Temporal Difference (TD) learning.

Features of the algorithms are:

- Stores Q-values for each state-action pair.
- Initialized dynamically when states are encountered.
- Update Q-values after every action using the formula: $Q(s,a) \leftarrow Q(s,a)+\alpha[r+\gamma Q(s',a')-Q(s,a)]$
- On-Policy Learning: Actions are selected using the same policy that the agent is evaluating (e.g., ϵ-greedy).
- Epsilon(ϵ)-Greedy Exploration: Balances exploration and exploitation during training.

Pseudocode

```
FOR each episode in range(num_episodes):
  IF episode is a multiple of 500:
    Reduce epsilon (epsilon = epsilon / 2)

  Initialize state (state = env.reset())
  Initialize the state in Q-table (initialize_state(state))
  Set total reward G = 0

  Choose action using epsilon-greedy policy (action = select_action(state))

  WHILE not terminal:
    Take the action in the environment
    Observe reward, next state, and done flag (next_state, reward, done = env.step(action))
    Initialize the next state in Q-table (initialize_state(next_state))

    Choose next action using epsilon-greedy policy (next_action = select_action(next_state))

    Compute TD target (td_target = reward + gamma * Q[next_state][next_action])
    Compute TD error (td_error = td_target - Q[state][action])
    Update Q-value for state-action pair (Q[state][action] = Q[state][action] + alpha * td_error)

    Update state and action for the next step (state = next_state, action = next_action)


  End WHILE

End FOR
```

Hyperparameters

The hyperparameters and their values used for the algorithm are:

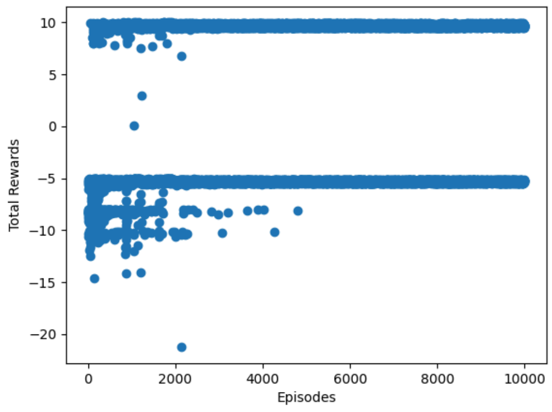
- Discount (Gamma) = 0.99
- Learning Rate (alpha) = 0.2
- Exploration threshold (Epsilon) = 0.5
- No. of episodes = 10,000

These values were adjusted for peak performance across several runs. Key takeaways include:

- A lower discount prevents the model from learning well. Since for this task, a shot from later states would be closer and is more likely to result in a goal, discounting future states heavily negatively impacts the algorithm.
- A stronger learning rate was necessary for the model to converge towards the best actions. Smaller learning rates such as 0.05 don't learn very well here as the q-values are less discernible.
- The threshold required a decay rate of 2 every 500 episodes in order to truly converge. Previously, a rate of 0.1 was used as a mostly greedy approach however, exploring early to find the best path and then stopping the exploration so that it only takes the best actions to score towards the end works best.
- A lower no. of episodes is not useful as for a sufficiently large field, all the states and paths would not be visited.

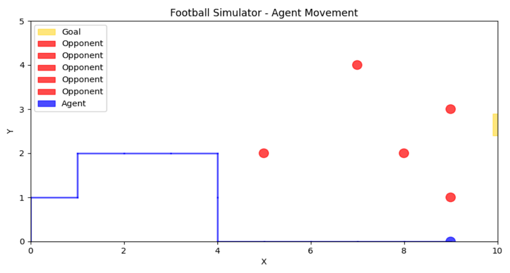
Results and Analysis

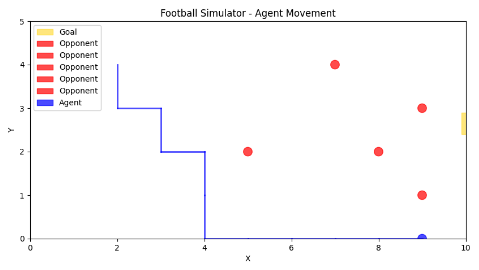
The average performance of the algorithm over 10 runs can be seen as below:



As seen above, the model converges from taking actions resulting in losing the ball (r= -10), or the shot being blocked (r=-8) to taking shots with a high chance to score.

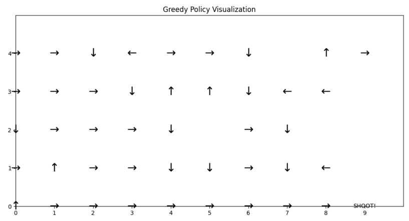
A few examples of the player in action starting from different states is shown below:





The model learns the best place to shoot for this scenario is at (9,0) resulting in an Expected goal(XG) of 0.7307417596432748. It devises a path accordingly, even diverging to avoid ball loss, to get to this state as quickly as possible.

The final learned policy for this example can be shown as below:
(Blank spaces are states with defenders or never visited as it would result in a ball loss)



Q-Learning

I implement the Q-learning algorithm that we learned in class on this MDP. Q-Learning is an off-policy algorithm that updates the Q-values based on the action that maximizes the Q-value for the next state, rather than the action actually taken.

Features of the algorithms are:

- Stores Q-values for each state-action pair, dynamically initialized when states are encountered.
- Q-Value Update Rule: $Q(s,a) \leftarrow Q(s,a)+\alpha[r+\gamma \max_{a'} Q(s',a')-Q(s,a)]Q(s, a)$
- Off-Policy Learning: Unlike SARSA, it uses the greedy action for the next state to calculate the TD target regardless of action actually taken.
- Epsilon-Greedy Exploration: ensures exploration during training.

Pseudocode

```
FOR each episode in range(num_episodes):
  IF episode is a multiple of 500:
    Reduce epsilon (epsilon = epsilon / 2)

  Initialize state (state = env.reset())
  Initialize the state in Q-table (initialize_state(state))

  Set total reward G = 0

  WHILE not done:
    Choose action using epsilon-greedy policy (action = select_action(state))

    Take the action in the environment
    Observe reward, next state, and done flag (next_state, reward, done = env.step(action))

    Update total return (G = G + reward)
    Initialize the next state in Q-table (initialize_state(next_state))

    Compute max Q-value for next state (max_q_next = max(Q[next_state][all actions]))
    Compute TD target (td_target = reward + gamma * max_q_next)
    Compute TD error (td_error = td_target - Q[state][action])
    Update Q-value for state-action pair (Q[state][action] = Q[state][action] + alpha * td_error)

    Update state for the next step (state = next_state)

    Add reward to total reward (total_reward += reward)

  End WHILE

End FOR
```

Hyperparameters

The hyperparameters and their values used for the algorithm are:
- Discount (Gamma) = 0.99
- Learning Rate (alpha) = 0.2

- Exploration threshold (Epsilon) = 0.5
- No. of episodes = 10,000

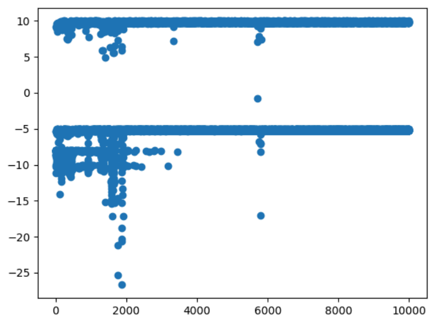
These values were adjusted for peak performance across several runs. Key takeaways include:

- A lower discount prevents the model from learning well. Since for this task, a shot from later states would be closer and is more likely to result in a goal, discounting future states heavily negatively impacts the algorithm.
- A stronger learning rate was necessary for the model to converge towards the best actions. Smaller learning rates such as 0.05 don't learn very well here as the q-values are less discernible.
- The threshold required a decay rate of 2 every 500 episodes in order to truly converge. Previously, a rate of 0.1 was used as a mostly greedy approach however, exploring early to find the best path and then stopping the exploration so that it only takes the best actions to score towards the end works best.
- A lower no. of episodes is not useful as for a sufficiently large field, all the states and paths would not be visited.

The hyperparameters were mostly kept consistent to gauge performance with other methods. For q-learning, the epsilon decay could be lessened from 0.5 to 0.3 to prevent exploration earlier as it is more unstable than Sarsa. The learning rate can also be increased to 0.25 if needed as it is capable of learning a little quicker.

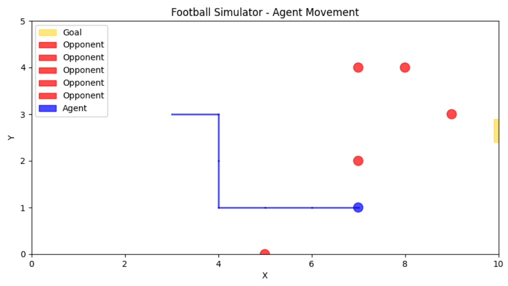
Results and Analysis

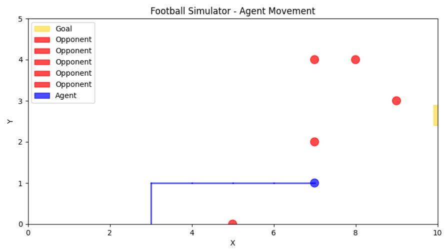
The average performance of the algorithm over 10 runs can be seen as below:



As seen above, the model converges similarly to avoiding bad actions of losing the ball and ends up taking high probability shots. However, compared to sarsa, the convergence is a little more unstable as the model still is seen exploring bad actions now and then quite late on despite learning quicker (approx 3900 episodes to Sarsa's 5400).

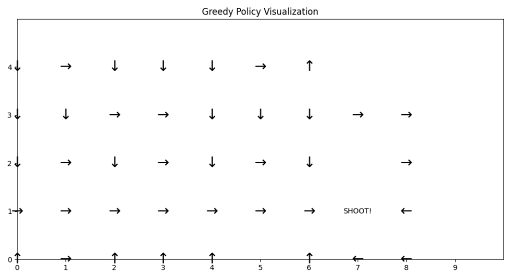
A few examples of the player in action starting from different states is shown below:





The model learns the best place to shoot for this scenario is at (7,1)) resulting in an Expected goal(XG) of XG: 0.6727998127341235.
It devises a path accordingly, even diverging to avoid ball loss, to get to this state as quickly as possible.

The final learned policy for this example can be shown as below:
(Blank spaces are states with defenders or never visited as it would result in a ball loss)



TRUE-ONLINE SARSA

Apart from the 2 methods covered in class, I also implemented a method that wasn't covered called True-Online Sarsa. This method is a better fit for my MDP than policy gradient methods like REINFORCE. Eligibility traces allow the agent to incorporate multi-step rewards efficiently, making it ideal for this football simulator environment where rewards (e.g., scoring or missing) may not be immediate.

Key features of this algorithm include:

- Use a table to store action-value pairs (s,a) initialized to zeros.
- Maintain a table for eligibility traces for each state-action pair.
- Tracks the degree of responsibility each state-action pair has for the TD error.
- Ensures backward propagation of the TD error across the trajectory.
- Update Q-values using:
  $\delta = r + \gamma Q(s', a') - Q(s, a)$.
  Adjust eligibility traces stored for each state-action pair with a trace vector z updated as: $z \leftarrow \lambda\gamma z + (1 - \alpha\lambda z^{\top} x(s,a))x(s,a)$ and use them to propagate the update:
  $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a) Q(s, a)$ where e(s,a) is the eligibility trace.
- Epsilon-Greedy policy: balances exploration and exploitation during learning.
- Its online nature provides feedback at each step, allowing me to observe how the agent learns while directly interacting with the environment.

Pseudocode

The chapter covered in the book incorporates a feature function x and a weight vector w. This formulation is designed for function approximation, where the Q-values are approximated using a linear combination of features: $\hat{q}(s,a,w) = w^{\top} x(s,a)$

The feature function x(s,a) is crucial for environments with large or continuous state-action spaces, where storing Q-values explicitly is infeasible. The current football simulator does not use a feature-based representation because the grid-based states can be handled with a Q-table. This approach is simpler for environments with limited states and actions.

A variant of this algorithm was implemented with a feature approximation however results were not performing well. The pseudocode of it is shown below:

```
player_pos = player_position from state
goal_pos = env.goal
distance_to_goal = Euclidean distance between player_pos and goal_pos
Initialize an empty list distances_to_opponents
FOR each opponent in opponent_positions:
    Calculate Euclidean distance between player_pos and opponent
    Add the distance to distances_to_opponents

IF distances_to_opponents is not empty:
    distance_to_nearest_opponent = Minimum value in distances_to_opponents
ELSE:
    distance_to_nearest_opponent = 0

action_encoding = Dictionary mapping actions to one-hot encoded lists
```

```
    Use action to retrieve the corresponding one-hot vector

    feature_vector = [
        Bias term (1.0),
        Negative distance_to_goal (to encourage moving closer to the goal),
        Negative distance_to_nearest_opponent (to encourage avoiding opponents)
    ] + action_encoding vector

Output:
    Return feature_vector
```

This was modified and instead implemented as the following: pseudocode:

```
FOR each episode in range(num_episodes):
    IF episode is a multiple of 500:
        Decay epsilon (epsilon = epsilon / 2)

    Reset environment and initialize state (state = env.reset())
    Choose initial action using epsilon-greedy policy (action = epsilon_greedy_policy(state))
    Clear eligibility traces (eligibility_traces.clear())
    Set total return G = 0

    WHILE not terminal:
        Take action, observe next state, reward, and done flag (next_state, reward, done = env.step(action))
        Choose next action using epsilon-greedy policy (next_action = epsilon_greedy_policy(next_state))
        Update total return (G = G + reward)

        Compute Q-values for current and next state-action pairs:
            q_current = get_q_value(state, action)
            q_next = get_q_value(next_state, next_action)

        Compute TD error:
            td_error = reward + gamma * q_next - q_current

        Update eligibility trace for the current state-action pair:
            eligibility_trace[state, action] += 1.0

        FOR each state-action pair (s, a) in eligibility_traces:
            Update Q-value:
                q_value = get_q_value(s, a)
                q_value += alpha * td_error * eligibility_trace[s, a]
                set_q_value(s, a, q_value)

            Decay eligibility trace:
                eligibility_trace[s, a] = gamma * lambda * eligibility_trace[s, a]

        Update current state and action for the next step:
            state = next_state
            action = next_action

End FOR
```

Hyperparameters

The hyperparameters and their values used for the algorithm are:

- Discount (Gamma) = 0.90
- Learning Rate (alpha) = 0.1
- Exploration threshold (Epsilon) = 0.5

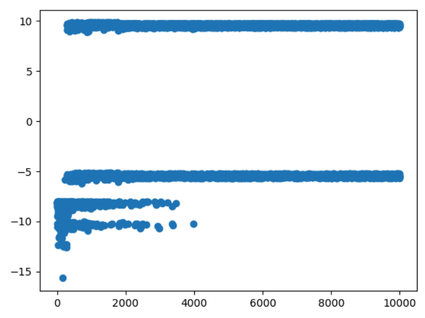
- No. of episodes = 10,000
- Trace Decay (Lambda) = 0.9

These values were adjusted for peak performance across several runs. Key takeaways include:

- Unlike the other two algorithms, A lower discount was usable for this algorithm. The assumption is that
- A lower learning rate was utilized here as the policy being learned here seemed to be more consistent. The actions chosen as a result are less erratic.
- Similarly as the other two, the threshold required a decay rate of 2 every 500 episodes in order to truly converge. Not using one does end up in the agent scoring goals, however it resorts to exploring non-high scoring shots as well due to the exploration parameter.
- A high no. of episodes is required as for a sufficiently large field, all the states and paths would not be visited.
- The trace decay was chosen as a high value. Lower values resulted in inefficient updates of the q-table.

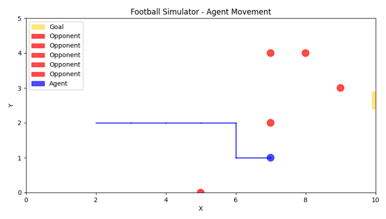
RESULTS AND ANALYSIS

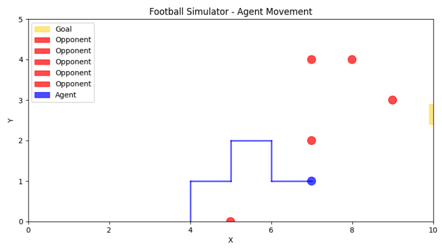
The average performance of the algorithm over 10 runs can be seen as below:

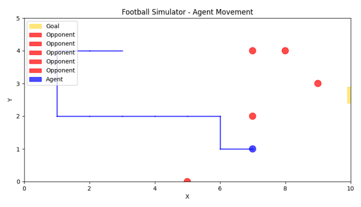


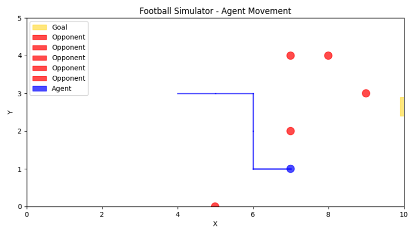
As seen above, the model converges earlier and much more smoothly than the other two algorithms.
It is able to identify bad reward paths quickly and adjust itself towards better and better actions.

A few examples of the player in action starting from different states is shown below:





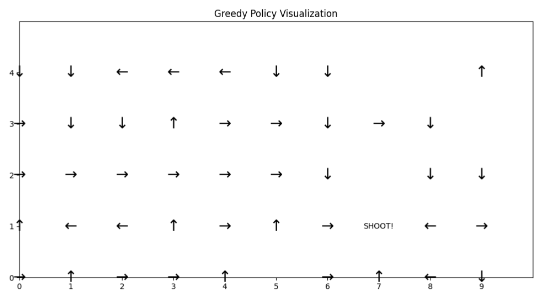




The model learns the best place to shoot for this scenario is at (7,1)) resulting in an Expected goal(XG) of XG: 0.6645898033750315.
However what's interesting to note is that the path that it devises every now and then is not ideal. This might be due to the fact that a proper function approximator hasn't been used in the algorithm which would help the agent understand transitions much better.

The final learned policy for this example can be shown as below:

(Blank spaces are states with defenders or never visited as it would result in a ball loss)



As can be seen, while the agent has managed to learn the best shooting location, the policy for the paths is not optimal.

CONCLUSION

All three algorithms are successfully able to converge and decide the best paths and positions to shoot from for the best chance at scoring a goal for the new designed football simulation MDP defined in this project.
While SARSA, relies heavily on exploration and learns well, Q-learning learns quicker however is more unstable. True-Online Sarsa offers the best balance in terms of performance and is more consistent. However, due to the complex nature of the eligibility trace updates, the policies learnt while effective and peform well, may not be optimal.