Soccer Game with Multi-agent Q-learning Algorithms

Huiwen Duan

I. Problem Statement

A special form of soccer game is the problem to solve as well as the environment to work in. It features 2 players over a game grid of 2(rows)*4(columns) cells. The ball may be possessed by either of the players. So the total number of states is 8(possible cells for player A) *7(possible cells for player B, since B cannot occupy the same cell as player A)*2(possession of the ball) = 112. The size of action space is 5(possible actions for player A)*5(possible actions for player B), yielding a total of 25 action combinations. The five actions N, S, E, W Stick. The first four actions indicate the directions of the action, whereas 'stick' means not to move. Each moving action will only help the player move one cell, unless the action ought to push the player out of the grid, in which case the player will stick.

In terms of reward, the two cells on the far left are goals for player A and the two cells on the far right are goals for player B. A player will be rewarded +100 if they bring the ball into their own goal but -100 if they bring the ball into their opponent's goal.The game is zero-sum, meaning the opponent's reward is the opposite of the reward earned by the player who enter the goal cells with a ball. For all other transitions, the reward is 0.

It is worth noting that our environment has no deterministic equilibrium policies, because for example in the state pictured below, B could be blocked by A regardless of the action B chooses. Therefore, it is important to find an algorithm that can converge to non-deterministic policies, namely foe-q and corr-q. Non-deterministic means given the states, the algorithm can predict more than one actions for the player to choose from.

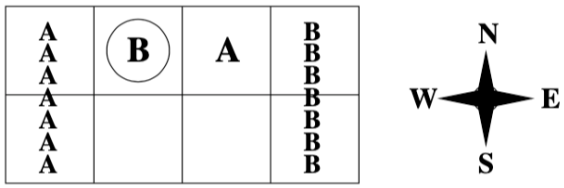

Fig. 1. Soccer Game at State S

II. Environment Implementation

I implemented my own soccer environment in a similar way as the Gym environments, with the following key functions:

- reset(): This function resets the environment, initialize the states, and returns the location of player A, location of player B, and who has ball. Both players are randomly placed in the middle non-goal cells, with the exception that no two players can occupy the same cell. Possession of the ball is randomly determined as well. In real world soccer games, no players start off within the area of goal. Moreover, there is a possibility of 1/4*3*2 that the reset will render the state S of interest, making sure that the game can experience this state at least 1 out of 24 times.
- step(actionA, actionB): This function will move the two players according to the actions inputted. The moving order of two players is randomly determined. The player will stick if the action pushes him out of the grid or causes him to collide with the other player. Collision will cause a change in the possession of balls. Specifically, if the action can cause the ball holder collide with the other player, the ball holder will pass the ball to the other player and stay at the original position. This function returns the next states, rewards for both players, and whether this specific game has ended.

There are also helper functions such as render(), which visualizes the current soccer playground, and sampleaction(), which randomly chooses an action from the actions space.

III. Algorithm Implementations

For all four algorithms, I implemented them according to Greenwald and Hall's table 1, as shown in figure 2. It is worth

For $t = 1$ to T

1. simulate actions $a_1, \ldots, a_n$ in state s
2. observe rewards $R_1, \ldots, R_n$ and next state s'
3. for $i = 1$ to n
   (a) $V_i(s') = f_i(Q_1(s'), \ldots, Q_n(s'))$
   (b) $Q_i(s, \vec{a}) = (1 - \alpha)Q_i(s, \vec{a}) + \alpha[(1 - \gamma)R_i + \gamma V_i(s')]$
4. agents choose actions $a'_1, \ldots, a'_n$
5. $s = s'$, $a_1 = a'_1$, $\ldots$, $a_n = a'_n$
6. decay α according to S

Fig. 2. Greenwald and Hall's multiagent Q-learning

A. Q-learning Implementation

Q learning only takes into consideration player A's action values, so we only need one Q table to represent them. It has 4 dimensions: 4 for the states and 1 for actions of A. I implemented exploration-exploitation trade-off with epsilon decay strategy so that the algorithm can explore more during the early learning episodes and exploit the visited territories more during the later state of training. I decayed epsilon by multiplying 0.999995 from a starting value of 1. I also decayed alpha by multiplying it with 0.999997 from a starting value of 1 to reach a floor of 0.05. The gamma I used is 0.9, the same as Greenwald. For each step, action value is updated according to the 3(b) step in Fig.2, with V being the maximum value in the Q table given the next states.

B. Friend-Q Implementation

Different from Q-learning, friend-Q adds in another player, and assumes that player B acts in the best interest of Player A. Every action player B takes is to maximize player A's value accumulation. Therefore, we can imagine player B to be a friend. We still need only one Q table, but including another dimension representing player B's actions. There are 25 action-pair values given the current state, and we want the objective V to be the maximum over these 25 values. I tweaked the exploration-exploitation strategy as well, to better manifest the fact that Player B is completely friendly. As decided randomly, player A can either take a random action or choose the action with the maximum value, and player B will always take the action that can maximum A's value return, even if this action minimizes his own returns. The alpha decay is the same as the previous algorithm.

C. Foe-Q Implementation

In Foe-Q learning, we assume Player B to completely hostile and will take whichever action that can minimize player A's value. The Q table is set up in a same way as Friend-Q, with 5 dimensions. The method of getting the objective V is different. In this fully competitive setting, player B will try to minimize over player A's actions before maximizing over his own actions. Player B learns how to defeat A based on how A acts and update its action values accordingly. Therefore, when getting to the objective V, we need to solve a mini-max problem. I used cvxpy as my linear programming tool. The variables are pi0 through pi4, constraints are set up in a way that for each action of B, the expected return for A is always smaller than v. Expected return is calculated by 1)multiplying the possibility of each A action with the value corresponding to action A and action B 2) summing the product over different actions of A. Each one of pi has to be bigger or equal to 0. The sum of the pi probabilities is 1. The objective is to maximize v. With all the constraints and objective, the maximized v is the object V value that we need to plug into the 3(b) equation in Fig. 2. In this algorithm, I let the agents choose actions randomly all the time and implemented the same alpha decay as the previous two algorithms.

D. uCE-Q Implementation

uCE-Q stands for utilitarian correlated equilibrium learning. Correlated Equilibria is a probability distribution over the joint space of actions (Greenwald and Hall, 2003).It takes into consideration that players actions can be dependent on each other.

A utilitarian CE tries to maximize the total combined reward of the two players. In a zero-sum game like our soccer environment, maximizing total rewards is the same as solving the minimax value functions. Therefore, theoretically uCE-Q will have the same results as foe-Q. Here we have two Q tables - one for player A and the other one for player B. Each table has 5 dimension - 3 for states, 1 for A's actions, and 1 for B's actions. The objective function for maximizing total rewards is shown in Fig. 3. In this algorithm, I let the agents choose actions randomly all the time and implemented the same alpha decay as the previous two algorithms.

The variable is the probability of all the action pairs, namely the joint distribution across players' actions, meaning P(A choosing S and B choosing S), PP(A choosing S and B choosing N), etc. This yields a total of 25 probabilities. The constraints

maximize the *sum* of the players' rewards:

$$\sigma \in \arg\max_{\sigma \in \text{CE}} \sum_{i \in I} \sum_{\vec{a} \in A} \sigma(\vec{a}) Q_i(s, \vec{a})$$

Fig. 3. uCE-Q Objective Function

are set up according to this logic: for each player, each action he chooses will have a larger or same expected value than any other actions. The expected value is calculated across opponent's actions. Therefore, each player has 5*4 = 20 constraints and there are a total of 20*2+25(all probabilities are greater than or equal to 0)+1(the sum of the probabilities is 1). The objective V for updating the Q table for player A is the maximized expected value across all action pairs and The objective V for player B is the maximized expected value across all action pairs. The two players always choose actions randomly.

I tested my uCE-Q solver with the Chicken-Dare game described in Greenwald's paper as well as the Wiki page([2]), and got $\begin{matrix} 0.5 & 0.25 \\ 0.25 & 0 \end{matrix}$ as the probability distribution and 10.5 as the total maximized value. This matches up with my classmate's results as posted on Piazza, which gives me confidence in my uCE-Q solver.

IV. Results

A. Q-learning

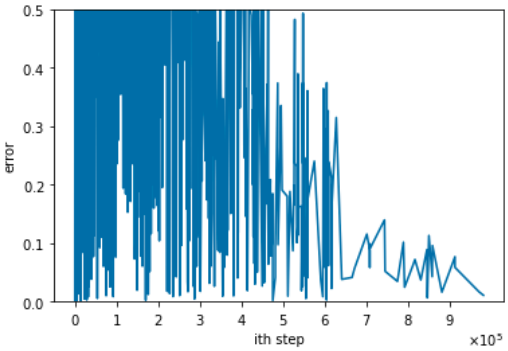

Fig. 4. Errors Trained With Q-learning

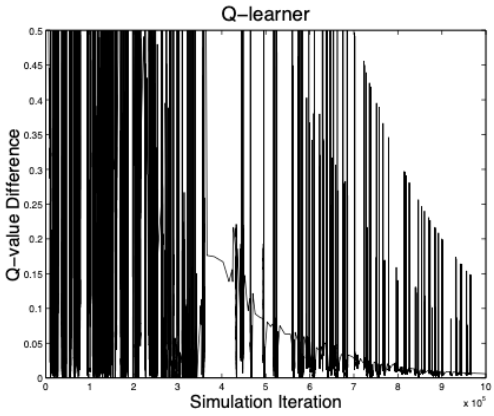

Fig. 5. Greenwald's Chart

Just as Greenwald's, my Q-learning never converged, although the amplitude of the error gets smaller and smaller. This is due to the decay in learning rate. According to Greenwald and Hall, "at all times, the amplitude of the oscillations in error values is as great as the envelope of the learning rate", because the errors we are plotting is proportional to the learning rate. Our charts don't look exactly the same, but the shape is similar, with the magnitude of error decreasing but never reaching 0. There are a couple reasons for the different plots. We might have different epsilon decay strategy and alpha decay strategy. Moreover, no random number generator is truly random, and the probability distribution behind our method of generating random actions might differ. Therefore, it should not be concerning that our plots don't look exactly the same.

B. Friend-Q

Similar to Greenwald's, my friend-q converges really fast and is even faster than the one in their paper. Besides reasons mentioned above, Greenwald and Hall might have had different plotting method because their Friend-Q chart looks really smooth, whereas my chart has a sharp decrease.

C. Foe-Q and uCE-Q

Foe-Q and uCE-Q have a really similar shape. As explained above, in a two player zero sum game, Foe-Q and uCE-Q should converge to the same equilibrium. They resemble the ones from Greenwald and Hall's paper as well. Both algorithms converged to a non deterministic equilibrium.

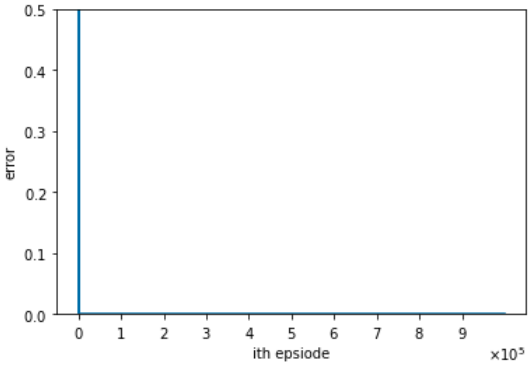

Fig. 6. Errors Trained With Friend-Q

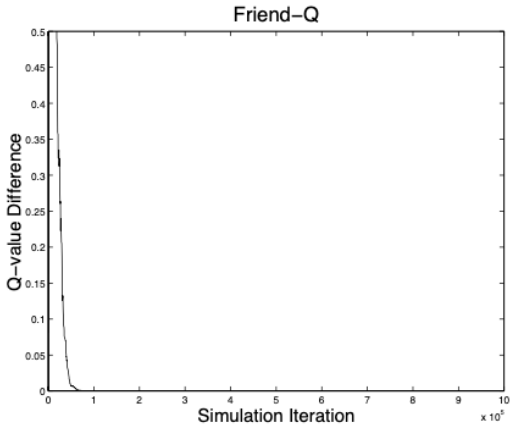

Fig. 7. Greenwald's Chart

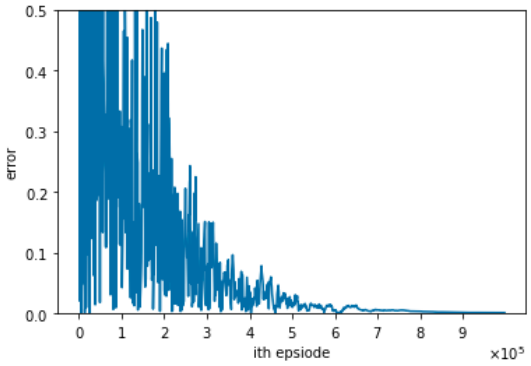

Fig. 8. Errors Trained With Foe-Q

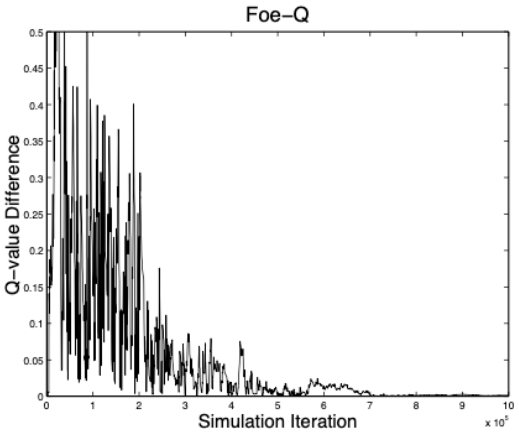

Fig. 9. Greenwald's Chart

V. Discussions and Future Directions

A. CVXPY vs CVXOPT

My implementations used CVXPY, which is the one introduced in HW6. However, multiple classmates as well as TA have recommended CVXOPT to substitute CVXPY. Is it suggested that CVXOPT can handle 0s more correctly. In my implementation of uCE-Q, CVXPT sometimes returns solutions that are really small and are almost zero. A classmate had similar issues on Piazza and was recommended to use CVXOPT, which is supposed to avoid the near zero values issues. For my implementation, I converted these values to 0 and normalized the rest of the probabilities. It is also suggested that CVXOPT has better solvers and enforce matrix representations for all the constraints, which can greatly reduce the run time. In the future, I want to try it.

B. Player Initialization

As I am writing this paper, I come up with a better way of initializing the players for the purpose of our experiments. In the future, I wan to initialize the game exactly as state S(Figure 1 state), the state that we are mostly interested in. Because the game is Markrov, the states prior to S are irrelevant and it is the states afterwards that shape the action-values for this state. Therefore we don't have to experience other states prior to visiting S. By initializing to S, we can make sure that S is visited in every episode of the game. This gives us more opportunity to update the action values with regard to S and give us more data points on the errors, which is what we need for testing and plotting the four algorithms. I implemented this to Friend-Q learning and got the same plot, and got way more data points(12280) as compared to the data points I got from the initialization described above (588), while the shape of the chart remains the same. However, the chart will more data points look different from Greenwald's chart, indicating that Greenwald potentially have used some smoothing functions for have used fewer data points. In the future I would like to apply this initialization strategy to testing the other three learning algorithms.

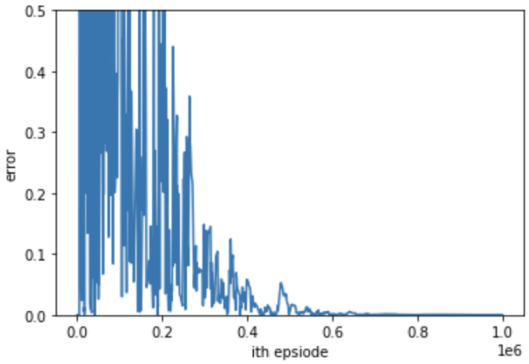

Fig. 10. Errors Trained With uCE-Q

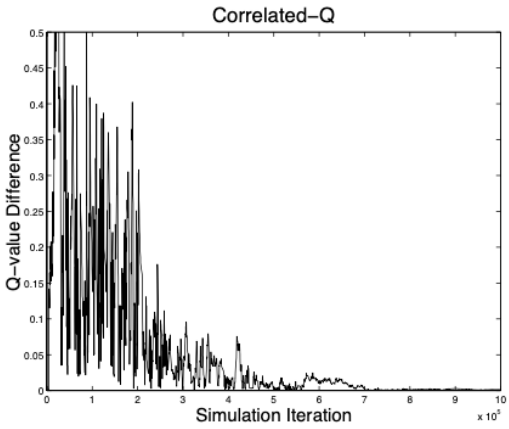

Fig. 11. Greenwald's Chart

C. 1 Million Iterations

At first I trained all my algorithms with 1 million episodes. For cor-Q, this took a really long time. Then I was inspired by Piazza and shortened it to 1 million steps instead. I wish Authors have shared their code or specified important parameters. I've found from both Project One and this project that a lot of researches don't pushlish the necessary details for reproducible results. As the person who's trying to reproduce the algorithms, it can take hours to guess the exact parameters and get to similar graphs/results.

REFERENCES

[1] Amy Greenwald, Keith Hall, and Roberto Serrano, Correlated Q-learning. In: ICML. Vol. 20. 1. 2003, p. 242.
[2] Wikipedia, The Free Encyclopedia, s.v. accessed April 19, 2021. Correlated equilibrium. https://en.wikipedia.org/wiki/Correlated$_equilibrium$