CS 7642 - Project 3 - Correlated-Q Learning

*Commit hash: 5c701959f26ae32bc4891f19d813e7666f7228cb

1st Jeremy Martinez
Dept. of Computer Science
Georgia Institute of Technology
Seattle, WA
jeremy.martinez@gatech.edu

***Abstract*—This is a recreation of the Soccer environment from Greenwald and Hall[1]. We apply four Q-learning algorithms to a two-person, zero sum Markov game. The four algorithms studied are Q-learning, Friend-Q, Foe-Q, and Correlated-Q learning. We observe that CE-Q performs most optimal.**

I. Introduction

Greenwald and Hall introduce Correlated-Q (CE-Q) learning, which is shown to outperform traditional Q-learning and Littman's Friend-Q and Foe-Q algorithms (FF-Q) in two person, zero sum Markov games. CE-Q generalizes both Nash-Q and FF-Q by finding minimax equilibria. This builds off of our solution in homework 6, in which we first solved for Nash equilibria using linear programming (LP). Using LP, we derive a probability distribution which the learner leverages to select an ϵ-greedy action. We apply this same strategy, using LP, in project 3, to solving the Soccer environment.

II. The Soccer Game

The soccer game we use to demonstrate these algorithms effectiveness was recreated from section 5 of Greenwald and Hall[1]. An image of the initial state of this game can be seen in figure 1. The environment is a two-player, zero-sum game. The game is played on a 8 cell (4x2) grid. Player A always starts in the top row, third column. Player B always start in the top row, second column. Player B always starts with the ball. The two cells to the left and the two to the right act as terminal states if a player enters that cell while in possession of the ball. The two left cells are player A's goal. The two to the right are player B's goal. If a player enters their goal with the ball, they receive a reward of +100 and the other player receives a reward of -100. If a player enters the opponents goal with the ball, they receive a reward of -100 and the other player receives a reward of +100. If the players move into the same cell, the player that acts second loses possession of the ball.

The action space is move up, right, down, left or stay put (do nothing). If a player attempts to move into a grid cell that is not on the board, their position will remain unchanged. For example, if player A attempts to move up from state s (in figure 1), and player B attempts to move down, then the next state (s') will have player A in the same position (1, 3) and player B will be one cell below (2, 2). Players execute their actions in random order.

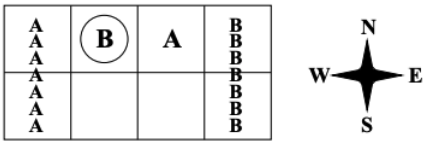

Fig. 1. Soccer game, state s

The Soccer environment was implemented as a Python class with three main functions: *get_state*, *simulate_action* and *render*. This structure was decided on after working with OpenAI's gym environment. The *get_state* function is trivial and simply returns a tuple of *player A's position, player B's position, ball possession*. The *render* function is also trivial and simply prints out (in the command line) a 4x2 grid show where the players are on the board and which player has the ball. The *simulate_action* actions as the transition function and can be seen in algorithm 1

A. Interfacing with the soccer environment

The four experiments conducted on the soccer environment all revolved around the same basic setup, which can be seen in algorithm 2. Where each algorithm varies is in the implementation of the function *select_epsilon_greedy_action*. The list of hyperparameter values can be see in table II-A. With these rates of decay, epsilon and alpha will reach their minimum at step 1,381,548. With a number of steps higher than this, neither the learning rate, nor the epsilon ever reach their minimum.

III. Markov Games

This soccer environment is an implementation of a Markov game - a stochastic game for which the probability transitions satisfy the Markov property[1]. The optimal value function at state s ($V^*(s)$) is defined by maximizing the Q-value ($Q^*(s, a)$) over the action space. The actions that maximize the Q-values also define the deterministic optimal policy π^*. This is illustrated in algorithm 3.

Markov games also define states slightly differently than the MDP we solved in project 2. Q-values are defined by stats and action vectors ($\overrightarrow{a}$), as opposed to state-action pairs. The Bellman equation changes only slightly as a result of this, adding n more dimensions, n being the number of players.

Algorithm 1: Soccer environment's transition function - *simulate_action*

Input Player A's action, player B's action
Output State, rewards vector, is_terminal
if *Invalid action taken* **then**
  return current state, [0, 0], is_terminal = False
end
Randomly select action order
Apply first player's action and update their position
if *Player A's position matches Player B's position* **then**
  if *Second player has possession* **then**
    Switch ball possession
  end
  return updated state, [0, 0], is_terminal = False
end
Apply second player's action and update their position
if *Player A reached A goal with ball* **then**
  return updated state, [+100, -100], is_terminal = True
end
else if *Player A reached B goal with ball* **then**
  return updated state, [-100, +100], is_terminal = True
end
else if *Player B reached B goal with ball* **then**
  return updated state, [-100, +100], is_terminal = True
end
else if *Player B reached A goal with ball* **then**
  return updated state, [+100, -100], is_terminal = True
end
return updated state, [0, 0], is_terminal = False

TABLE I
LIST OF HYPERPARAMETERS FOR TRAINING AGENT

γ	0.9
ϵ	1
ϵ-decay	0.000005
ϵ-min	0.001
α	1
α-decay	0.001
α-min	0.000005
steps	1000000

Since there are only two players, this only increases the dimension of the Q-value table by 1.

A. Nash equilibrium

With this action vector, we can now approach optimizing this differently. We look for Nash equilibriums between the two players action spaces in order to optimize the overall reward. It helps to divide the Q-value dimension space into two parts for intuition.

Each Q-value is comprised of 5 dimensions: player A position, player B position, ball possession, player A action, player B action. We will dissect this state space in two parts. First, player position and ball possession are a reflection of the soccer environment, and so it requires no decision from the agent. Player position can have a value of 1..8. Ball possession is a binary value: 0 indicates player B has the ball and 1 indicates player A has the ball.

Algorithm 2: Training an agent to learn the soccer environment

Init Q-value table
Init list of errors $\doteq error\_list$
for *N number of steps* **do**
  Init soccer environment $\doteq env$
  while *True* **do**
    S $\doteq env.get\_state()$
    $\overrightarrow{a} \doteq select\_epsilon\_greedy\_action$
    S', R, done = env.simulate_action($\overrightarrow{a}$)
    record Q-value at state s $\doteq before$
    update Q-values using Bellman's equation
    record Q-value at state s $\doteq after$
    decay epsilon and learning rates
    error_list.append(before - after)
  end
end
return error_list

$\Pi^*(s) \in arg\max_{\overrightarrow{a} \in A_s} Q^*(s, a)$

For the action vector portion of the state space, each action can have a value of 1..5. We can visualize this as a 5x5 grid, and discuss what an equilibrium looks like similarly to how the lecture videos discuss prisoners dilemma. In table III-A, we see the Q-values across the action vector. This table have a Nash equilibrium for the action vector [down, down], since no player can increase their score given the other players action is fixed. This would not be the case for the action vector [right, left]. For this action vector, player B can increase their reward (from -100 to 0) be taking any other action. A similar argument can be made for the action vector [do nothing, up].

B. Correlated-Q learning

A correlated equilibrium is also a probability distribution, like Nash, with respect to one another's probabilities with added constraints. An important distinction between these two is that this correlated equilibrium can be computed easily via linear programming. In Greenwald and Hall, they illustrate these constraints with an example, using the game "chicken." Chicken is a two-player, two-action general-sum game. The reward structure can be seen below in figure 2. The probability

TABLE II
ACTION VECTOR STATE-SPACE

0, 0	0, 0	0, 0	0, 0	0, 0
0, 0	0, 0	0, 0	100, -100	0, 0
0, 0	0, 0	60, 40	0, 0	0, 0
0, 0	0, 0	0, 0	0, 0	0, 0
-100, 100	0, 0	0, 0	0, 0	0, 0

	L	R
T	6,6	2,7
B	7,2	0,0

Fig. 2. Chicken rewards

$$-1\pi_{TL} + 2\pi_{TR} \geq 0 \qquad -1\pi_{TL} + 2\pi_{BL} \geq 0$$
$$1\pi_{BL} - 2\pi_{BR} \geq 0 \qquad 1\pi_{TR} - 2\pi_{BR} \geq 0$$

Fig. 3. Chicken rationality constraints

distribution of this action-vector always adds up to 1, and they are all non-negative. Further, rationality constraints are applied (see below in figure 3). For intuition, we will talk through the first (upper-right) constraint from figure 3: the probability of taking action L given action T (π_{TL}) is less than or equal to half of taking action R given action T (π_{RL}): $2\pi_{\text{TR}} \geq \pi_{\text{TL}}$.

IV. Q-Learning

The off-policy Q-learning implementation here serves as our base-case to build from. The action selection here simply takes the max action from Q-value action space, relative to the Q-values of the player at hand. This essentially looks like the following: $\max_{a \in A_s} Q(Aposition, Bposition, ballpossession, a)$.
This learner does not converge, as we can see in figure 4. The Q-value error acts sporadically for nearly all steps leading up to 1,000,000 steps. The line deceivingly looks like it converges toward the end. However, this is just simply the actions are being taken more deterministically.

As we can see here, this learner differs from the other 3 in that it does not evaluate the opponents action space in it's Q-values. For this reason, it makes no attempt to optimize it's Q-values based off of the other players available actions.

V. Friend-Q

Greenwald and Hall implement Michael Littman's Friend-Q and Foe-Q learners here for comparison[2].

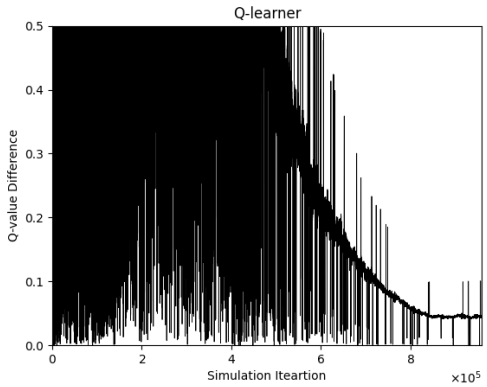

Fig. 4. Q-learner performance measured by Q-value error

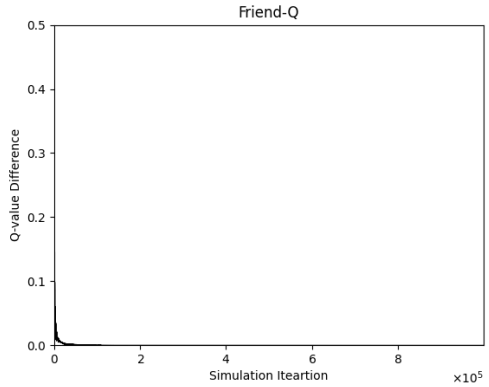

Fig. 5. Friend-Q performance measured by Q-value error

The Friend-Q implementation is similar to our base Q-learner. The Q-value dimensions, as we see in algorithm 4, takes the max Q-value with consideration to both players actions. This will look like the following: $\max_{a_1 \in A_1, a_2 \in A_2} Q(Aposition, Bposition, ballpossession, a_1, a_2)$. Figure 5 shows that the policy fails to converge as well. The Q-value errors falls to zero at around 80,000 steps.

Algorithm 3: Friend-Q action selection[2]

$V_i(s) = \max_{a_1 \in A_1, a_2 \in A_2} Q_i(s, a_1, a_2)$

VI. Foe-Q

Foe-Q attempts to find an adversarial equilibrium. It is a slight modification of Friend-Q in that it attempts to maximize the Q-value over it's own action, while minimizing the Q-value over the opponents action. We see this change reflected in algorithm 5. This is when we first introduce cvxopt to solve this equilibrium with linear programming. This is an application of maximin (opposite of minimax from lecture).

The maximin linear programming portion of the *select_epsilon_greedy_action* function provides us with a probability distribution which we use to select an action. This will look like the following (using numpy): *np.random.choice(np.arange(5), 1, p=probability_distribution)*

After training the agent on Foe-Q for 1,000,000 steps, we can observe the Q-values for Q[2][1][1] (state s from figure 1) in figure 7. This shows that it is most optimal for player A to take action 4. It also shows that Player A's expected reward goes down if player B takes action right and down. This follows intuition since we can expect a player B to have a direct line of sight to their goal in both of these scenarios. We also expect moving left to have a significant higher expected value since it is attempting to steal possession of the ball from player B.

Algorithm 4: Foe-Q action selection

$V_1(s) = \max_{\overrightarrow{a1} \in A_1} \max_{\overrightarrow{a2} \in A_2} Q_i(s, \overrightarrow{a1}, \overrightarrow{a2}) = -V_2(s)$

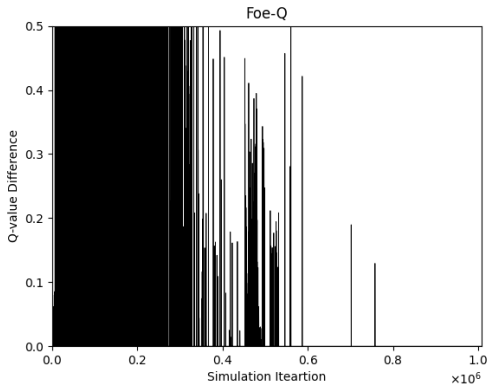

Fig. 6. Foe-Q performance measured by Q-value error

VII. Correlated-Q

The correlated-Q (CE-Q) learner extends our linear programming portion of the learner from Foe-Q. However, outside of computing this probability distribution, the rest of the algorithm looks identical. Greenwald and Hall discuss four approaches to CE-Q.

For our CE-Q learner, we implemented uCE-Q. We can see the algorithm, at a high level, in algorithm 6. The linear programming helper is attempting to maximize the sum of both of the player's rewards. At first thought, from a particle perspective, this didn't exactly make the most sense to me in a zero-sum game. In a real world application, it does not make sense to attempt to maximize your opponents reward, since this indirectly means minimizing your reward (zero sum). However, we can see after 1,000,000 steps, the Q-values offer much more insight to what the expected reward may be compared to Foe-Q.

See the Q-value distribution from CE-Q in figure 12. Let's dissect these Q-values and see if they provide more insight than those from Foe-Q. If player A knows what move player B will make, then the values that yield the highest value are the follow action vectors:

```
array([[-25.52653607, -26.24250971, -40.25930925, -20.68574094, -26.23043632], # Sum = -138.94
       [ 27.53250604, -69.85959089, -10.68065794,  -7.95048384, 27.22715862],  # Sum = -33.73
       [-38.25743722, -33.27514472, -27.64361097, -57.09724923, -37.82085874], # Sum = -194.09
       [100.00592113,  95.94951209,  90.66454666, 100.09507285, 100.00918954], # Sum = 486.72
       [-14.42339271, -24.91098368, -15.65804574, -22.95780269, -21.8556865 ], # Sum = -99.80
])
```

Fig. 7. Player 1 Q-value at state s from Foe-Q

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

Fig. 8. Utilitarian CE-Q: maximize the sum of the players' rewards

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

Fig. 9. Egalitarian CE-Q: maximize the minimum of the players' rewards

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

Fig. 10. Republican CE-Q: maximize the maximum of the players' rewards

- Player A: up, Player B: left $=>$ **100**
- Player A: right, Player B: left $=>$ **93**
- Player A: down, Player B: left $=>$ **93**
- Player A: left, Player B: left $=>$ **100**
- Player A: up, Player B: left $=>$ **100**

This is interesting, and offers some intricate insight that Foe-Q-values does not. It is stating that, under any situation, as long as player B goes left, it will result in a high reward for player A. Also, it states that if player A moves left, but player B moves right, it could lead to a very low reward. Foe-Q said otherwise (expected reward of 90). This makes a lot of sense, since this action vector would result in both players switching places, placing player B closer to their goal with the ball.

Figure 13 shows that CE-Q converges after about 800,000 steps, since each games error approaches zero, with an optimal policy. Greenwald and Hall point out that CE-Q eventually converges to a minimax equilibrium. We see this in our own results when reproducing these experiments.

VIII. Recapitulating Experiment Results

We were able to get fairly close to the figures from Greenwald and Hall. Our results of Friend-Q are a little off in that the error approaches zero a lot quicker. The basic Q-learner is sporadic and matches the results fairly well. CE-Q and Foe-Q also converge similarly to Greenwald and Hall's results.

Reproducing these experiments were useful for gaining insight to how LP can help find these equilibriums. Tinkering with the Q-values, setting up the state-action space, and especially developing the Soccer environment all reflected useful real-world applications. Developing the soccer environment after having worked with the OpenAI environment was

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

Fig. 11. Libertarian CE-Q: maximize the maximum of each individual payer i's rewards

```
array([[-28.14977222,  26.675575  , -40.68222627, 100.01305744, -29.5049735 ], # Sum = 28.35
       [-25.58299327, -68.15270074, -32.35502659,  93.03058547, -28.52024843], # Sum = -61.58
       [-32.08746611, -27.51065476, -29.50462846,  93.42415486, -39.79222159], # Sum = -35.47
       [-27.81099399,  -1.39803943, -54.32455873, 100.01192294, -27.14936013], # Sum = -10.67
       [-26.22340824,  25.78830839, -41.03535721, 100.01368714, -29.82234738], # Sum = 28.72
])
```

Fig. 12. Player 1 Q-value at state s from CE-Q

Algorithm 5: Correlated-Q action selection

$V_i(s) \in CE_i(Q_1(S), ..., Q_n(s))$

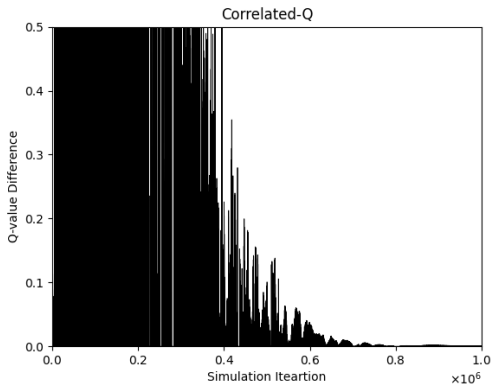

Fig. 13. Correlated-Q performance measured by Q-value error

interesting to see how important it is to set up both sides of the learner.

IX. Pitfalls & Problems

One major issue when developing this was the runtime of the algorithm. The script takes several hours to complete, which made testing these algorithms very difficult. I typically tested this by only running it at about 100,000 - 300,000 steps in order to save time, then extrapolate to get an idea of how it will perform over all 1,000,000 steps. That way, when I eventually did run it at full length, I was able to do so with decent certainty that it would match the results of Greenwald and Hall.

One other issue was knowing how to properly unit test my LP helper function. To be able to isolate and unit test this, as well as the ϵ-greedy action selection. I was able to unit test the soccer env pretty well. This was the first thing I did when starting out this project. This way, it was a tested and trusted portion of the code, and I didn't have to worry about this while working on the four Q-learner algorithms

X. Further Research

Given more time, I would have liked to experiment with a more rich reward structure. Developing a more elaborate environment gives you the ability to manipulate how the agent treats it's environment. Shaping these rewards in different ways will show insight on how you can get convergence faster. This was eluded to by Charles and Michael in the lecture series when they suggested that one easy way to find an equilibrium is to simply changes the rewards. This jokingly feels a little like cheating the system. However, when you build the system, you have the liberty to do this.

I would also like to explore and read more about the cvxopt library. The more experience and knowledge one gets around this API, the more one can leverage more intricate features. The linear algebra of the matrices and the solvers is also something that I would like to grasp a bit further.

References

[1] Amy Greenwald, Keith Hall, and Roberto Serrano. Correlated Q-learning. In: ICML. Vol. 20. 1. 2003, p. 242.

[2] M. Littman. Friend or foe Q-learning in general-sum Markov games. In Proceedings of Eighteenth International Conference on Machine Learning, pages 322328, June 2001.