

Homework 5: Reinforcement Learning and Game Theory

CMU 15-780: Graduate AI (Spring 2023)

Out: Apr 10, 2023
Due: Apr 24, 2023 11:59 pm

Instructions

Collaboration Policy

You may discuss assignments with other students as you work through them, but writeups must be done alone. No downloading or copying of code or other answers is allowed. If you use a string of at least 5 words from some source, you must cite the source. Do not use any third-party libraries except those explicitly stated in this handout.

Submission

You should submit three files via Gradescope: `q_learning.py` and `cfr.py` for the programming portion, and a PDF for your answers to the writtens. Your completed functions will be autograded by running through several test cases (not just the ones supplied with starter code).

You have 8 late days for homeworks over the semester, and can use at most 3 for one homework.

1 Q-Learning [50 pts]

This homework consists of both written and programming questions in both parts.

In this problem you will implement a basic version of tabular Q-learning to train an agent to play CartPole (https://www.gymnasium.dev/environments/classic_control/cart_pole/). Install OpenAI Gym with `pip install gym`. Here, the agent controls a cart on a frictionless 1-D track. A pole is attached to the cart by a single unmovable hinge. The pole starts in the upright position, and the goal is to keep the pole balanced by moving the cart left and right. The state of the CartPole environment is described by four parameters $[x, v, \theta, \omega]$ where x is the position of the cart, v is the velocity of the cart, θ is the angle of the pole, and ω is the angular velocity of the pole. Each episode consists of at most 500 time steps, and a reward of 1 is obtained for each time step where the pole is upright. The episode ends if the cart leaves the track ($x \notin (-2.4, 2.4)$) or if the pole falls ($\theta \notin (-.2095, .2095)$).

Preparing the environment. Since we are implementing tabular Q-learning, we need to operate on a suitable discretization of the state space. This is achieved by discretizing the range of states into evenly-spaced buckets (see the starter code for details).

1. [15 points] Implement `create_discretization`, `get_discretized_state`, and `create_q_table` in `q_learning.py`.

Q-Learning. You will now implement the Q-learning algorithm and use the epsilon-greedy policy to select actions. We'll train our agent over 10000 episodes.

2. [15 points] Implement `epsilon_greedy` and `q_update` in `q_learning.py`.
3. [5 points] (*Written*) Set `epsilon = 1` and execute `cartpole.py`. What action-selection policy does `epsilon = 1` correspond to? What is the average reward obtained by the agent? (A rough answer

rounded to the nearest tens place is okay.)

4. [15 points] (*Written*) Play around with the parameter `epsilon` in `cartpole.py` until you see average scores that are at least 195 along the 10000 episodes. Plot the average score for every 100th episode and upload your plot, along with an explanation of the parameters you chose, to Gradescope. You may wish to implement some form of *epsilon decaying*, where `epsilon` starts at 1 and decays gradually to 0 over future episodes. Feel free to modify the learning rate and discount as well.

2 Counterfactual Regret Minimization [50+ points]

In this problem, you will implement the CFR regret minimizer for sequence-form decision problems.

You will run your CFR implementation on three games: rock-paper-superscissors (a simple variant of rock-paper-scissors, where beating paper with scissors gives a payoff of 2 instead of 1) and two well-known poker variants: Kuhn poker [Kuhn, 1950] and Leduc poker [Southey et al., 2005]. A description of each game is given in the zip of this homework, according to the format described in Section 2.1.

2.1 Format of the game files

Each game is encoded as a json file with the following structure.

- At the root, we have a dictionary with three keys: `decision_problem_p11`, `decision_problem_p12`, and `utility_p11`. The first two keys contain a description of the tree-form sequential decision problems faced by the two players, while the third is a description of the bilinear utility function for Player 1 as a function of the sequence-form strategies of each player. Since both games are zero-sum, the utility for Player 2 is the opposite of the utility of Player 1.

- The tree of decision points and observation points for each decision problem is stored as a list of nodes. Each node has the following fields

`id` is a string that represents the identifier of the node. The identifier is unique among the nodes for the same player.

`type` is a string with value either `decision` (for decision points) or `observation` (for observation points).

`actions` (only for decision points). This is a set of strings, representing the actions available at the decision node.

`signals` (only for observation points). This is a set of strings, representing the signals that can be observed at the observation node.

`parent_edge` identifies the parent edge of the node. If the node is the root of the tree, then it is `null`. Else, it is a pair (`parent_node_id`, `action_or_signal`), where the first member is the `id` of the parent node, and `action_or_signal` is the action or signal that connects the node to its parent.

`parent_sequence` (only for decision points). Identifies the parent sequence p_j of the decision point, defined as the last sequence (that is, decision point-action pair) encountered on the path from the root of the decision process to j .

Remark 1. The list of nodes of the tree-form sequential decision process is given in top-down traversal order. The bottom-up traversal order can be obtained by reading the list of nodes backwards.

- The bilinear utility function for Player 1 is given through the payoff matrix \mathbf{A} such that the (expected) utility of Player 1 can be written as

$$u_1(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\top \mathbf{A} \mathbf{y},$$

where \mathbf{x} and \mathbf{y} are sequence-form strategies for Players 1 and 2 respectively. We represent \mathbf{A} in the file as a list of all non-zero matrix entries, storing for each the row index, column index, and value. Specifically, each entry is an object with the fields

`sequence_pl1` is a pair (`decision_pt_id_pl1`, `action_pl1`) which represents the sequence of Player 1 (row of the entry in the matrix).

`sequence_pl2` is a pair (`decision_pt_id_pl2`, `action_pl2`) which represents the sequence of Player 2 (column of the entry in the matrix).

`value` is the non-zero float value of the matrix entry.

Example: Rock-paper-superscissors In the case of rock-paper-superscissors the decision problem faced by each of the players has only one decision points with three actions: playing rock, paper, or superscissors. So, each tree-form sequential decision process only has a single node, which is a decision node. The payoff matrix of the game¹ is

$$\begin{matrix} & \begin{matrix} r & p & s \end{matrix} \\ \begin{matrix} r \\ p \\ s \end{matrix} & \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -2 \\ -1 & 2 & 0 \end{pmatrix} \end{matrix}.$$

So, the game file in this case has content:

```
{
  "decision_problem_pl1": [
    {"id": "d1_pl1", "type": "decision", "actions": ["r", "p", "s"],
      "parent_edge": null, "parent_sequence": null}
  ],
  "decision_problem_pl2": [
    {"id": "d1_pl2", "type": "decision", "actions": ["r", "p", "s"],
      "parent_edge": null, "parent_sequence": null}
  ],
  "utility_pl1": [
    {"sequence_pl1": ["d1_pl1", "r"], "sequence_pl2": ["d1_pl2", "p"], "value": -1},
    {"sequence_pl1": ["d1_pl1", "r"], "sequence_pl2": ["d1_pl2", "s"], "value": 1},
    {"sequence_pl1": ["d1_pl1", "p"], "sequence_pl2": ["d1_pl2", "r"], "value": 1},
    {"sequence_pl1": ["d1_pl1", "p"], "sequence_pl2": ["d1_pl2", "s"], "value": -2},
    {"sequence_pl1": ["d1_pl1", "s"], "sequence_pl2": ["d1_pl2", "r"], "value": -1},
    {"sequence_pl1": ["d1_pl1", "s"], "sequence_pl2": ["d1_pl2", "p"], "value": 2}
  ]
}
```

For this assignment, there is no need to worry about constant-factor speedups that would be given, e.g., by vectorizing where possible. Also, note that we have performed the conversion from extensive form to bilinear saddle-point form, that is, the form $\max_{\mathbf{x}} \min_{\mathbf{y}} \mathbf{x}^\top \mathbf{A} \mathbf{y}$, for you.

2.2 Learning to best respond

Let \mathcal{X} and \mathcal{Y} be the sequence-form strategy polytopes corresponding to the tree-form sequential decision problems faced by Players 1 and 2 respectively. A good smoke test when implementing regret minimization algorithms is to verify that they learn to best respond. In particular, you will verify that your implementation of CFR applied to the decision problem of Player 1 learns a best response against Player 2 when Player 2 plays the *uniform* strategy, that is, the strategy that at each decision points picks any of the available actions with equal probability.

Let $\mathbf{y} \in \mathcal{Y}$ be the sequence-form representation of the strategy for Player 2 that at each decision point selects each of the available actions with equal probability. When Player 2 plays according to that strategy, the utility vector for Player 1 is given by $\mathbf{u} := \mathbf{A} \mathbf{y}$, where \mathbf{A} is the payoff matrix of the game.

¹A Nash equilibrium of the game is reached when all players play rock with probability 1/2, paper with probability 1/4 and superscissors with probability 1/4. Correspondingly, the game value is 0.

For each of the three games, suppose that you take your CFR implementation for the decision problem of Player 1, and let it output strategies $\mathbf{x}^{(t)} \in \mathcal{X}$ while giving as feedback at each time t the same utility vector \mathbf{u} . As $T \rightarrow \infty$, the average strategy

$$\bar{\mathbf{x}}^{(T)} := \frac{1}{T} \sum_{t=1}^T \mathbf{x}^{(t)} \in \mathcal{X}$$

will converge to a best response to the uniform strategy \mathbf{y} , that is,

$$\lim_{T \rightarrow \infty} (\bar{\mathbf{x}}^{(T)})^\top \mathbf{A} \mathbf{y} = \max_{\hat{\mathbf{x}} \in \mathcal{X}} \hat{\mathbf{x}}^\top \mathbf{A} \mathbf{y}.$$

If the above doesn't happen empirically, something is wrong with your implementation.

1. [10 points] (Warm-up) Implement the functions `expected_utility_pl1` and `uniform_sf_strategy`, following the directions in `cfr.py`.
2. [10 points] Implement the classes `RegretMatching` and `Cfr`.

If implemented correctly, `cfr_test.py` should now correctly compute best response values against the uniform random strategy in all three games.

2.3 Learning a Nash equilibrium

Now that you are confident that your implementation of CFR is correct, you will use CFR to converge to Nash equilibrium using the self-play idea described in lecture and recalled next.

The idea behind using regret minimization to converge to Nash equilibrium in a two-player zero-sum game is to use *self play*. We instantiate two regret minimization algorithms, $\mathcal{R}_\mathcal{X}$ and $\mathcal{R}_\mathcal{Y}$, for the domains of the maximization and minimization problem, respectively. At each time t the two regret minimizers output strategies $\mathbf{x}^{(t)}$ and $\mathbf{y}^{(t)}$, respectively. Then, they receive as feedback the vectors $\mathbf{u}_\mathcal{X}^{(t)}, \mathbf{u}_\mathcal{Y}^{(t)}$ defined as

$$\mathbf{u}_\mathcal{X}^{(t)} := \mathbf{A} \mathbf{y}^{(t)}, \quad \mathbf{u}_\mathcal{Y}^{(t)} := -\mathbf{A}^\top \mathbf{x}^{(t)}, \quad (1)$$

where \mathbf{A} is Player 1's payoff matrix.

We summarize the process pictorially in Figure 1.

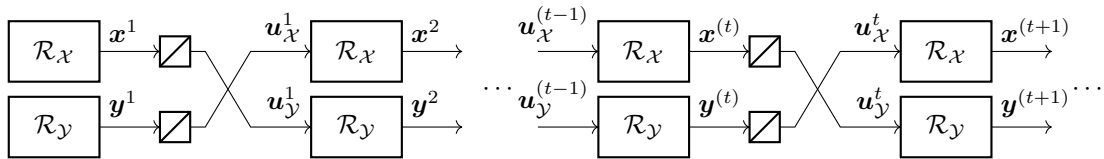


Figure 1: The flow of strategies and utilities in regret minimization for games. The symbol \boxtimes denotes computation/construction of the utility vector.

A well known folk theorem establish that the pair of average strategies produced by the regret minimizers up to any time T converges to a Nash equilibrium, where convergence is measured via the *saddle point gap*

$$0 \leq \gamma(\mathbf{x}, \mathbf{y}) := \left(\max_{\hat{\mathbf{x}} \in \mathcal{X}} \{\hat{\mathbf{x}}^\top \mathbf{A} \mathbf{y}\} - \mathbf{x}^\top \mathbf{A} \mathbf{y} \right) + \left(\mathbf{x}^\top \mathbf{A} \mathbf{y} - \min_{\hat{\mathbf{y}} \in \mathcal{Y}} \{\mathbf{x}^\top \mathbf{A} \hat{\mathbf{y}}\} \right) = \max_{\hat{\mathbf{x}} \in \mathcal{X}} \{\hat{\mathbf{x}}^\top \mathbf{A} \mathbf{y}\} - \min_{\hat{\mathbf{y}} \in \mathcal{Y}} \{\mathbf{x}^\top \mathbf{A} \hat{\mathbf{y}}\}.$$

A point $(\mathbf{x}, \mathbf{y}) \in \mathcal{X} \times \mathcal{Y}$ has zero saddle point gap if and only if it is a Nash equilibrium of the game.

Theorem 2. Consider the self-play setup summarized in Fig. 1, where $\mathcal{R}_\mathcal{X}$ and $\mathcal{R}_\mathcal{Y}$ are regret minimizers for the sets \mathcal{X} and \mathcal{Y} , respectively. Let $R_\mathcal{X}^{(T)}$ and $R_\mathcal{Y}^{(T)}$ be the (sublinear) regret cumulated by $\mathcal{R}_\mathcal{X}$ and $\mathcal{R}_\mathcal{Y}$, respectively, up to time T , and let $\bar{\mathbf{x}}^{(T)}$ and $\bar{\mathbf{y}}^{(T)}$ denote the average of the strategies produced up to time T , that is,

$$\bar{\mathbf{x}}^{(T)} := \frac{1}{T} \sum_{t=1}^T \mathbf{x}^{(t)}, \quad \bar{\mathbf{y}}^{(T)} := \frac{1}{T} \sum_{t=1}^T \mathbf{y}^{(t)}.$$

Then, the saddle point gap $\gamma(\bar{\mathbf{x}}^{(T)}, \bar{\mathbf{y}}^{(T)})$ of $(\bar{\mathbf{x}}^{(T)}, \bar{\mathbf{y}}^{(T)})$ satisfies

$$\gamma(\bar{\mathbf{x}}^{(T)}, \bar{\mathbf{y}}^{(T)}) \leq \frac{R_{\mathcal{X}}^{(T)} + R_{\mathcal{Y}}^{(T)}}{T} \rightarrow 0 \quad \text{as } T \rightarrow \infty.$$

3. [20 points] Implement the function `run_cfr`, which takes in a game and solves it by running CFR on the game for the specified number of iterations.

★ Hint: represent vectors on $\mathbb{R}^{|\Sigma|}$ (including the sequence-form strategies output by CFR and utility vectors given to CFR) in memory as dictionaries from sequences (tuples (`decision_point_id`, `action`)) to floats.

★ Hint: to compute the saddle-point gap, feel free to use the function `gap(game, strategy_p11, strategy_p12)` provided in the starter code.

★ Hint: the saddle point gap should be going to zero. The expected utility of the average strategies in rock-paper-superscissor should approach the value 0. In Kuhn poker it should approach -0.055 . In Leduc poker it should approach -0.085 .

4. (*Written*) In the game Kuhn poker, inspect the returned equilibrium strategies of both P1 and P2 after 1000 iterations of CFR, and answer the following questions. (You may use whatever method you wish to perform this inspection)

- (a) [5 points] Jack is the worst starting card in Kuhn poker. In the strategy profile computed by CFR, with what probability does Player 1 fold (P1:f) while having a Jack if Player 2 bets (decision point /C:J/P1:c/P2:r)?

In a sentence or two, explain why this makes sense.

- (b) [5 points] Since Jack is the worst starting card, betting with a Jack is a bluff, since the player who bets is guaranteed to lose if the bet is called. In the strategy profile computed by CFR, with what probability does Player 2 bet (P2:r) while having a Jack if Player 1 checks (decision point C:?J/P1:c)?

In a sentence or two, explain why this makes sense.

2.4 Extra credit: DCFR and Predictive CFR+

This section is extra credit.

To achieve better performance in practice when learning Nash equilibria in two-player zero-sum games, people often make the following modifications to the setup of the previous subsection.

- Instead of regret matching, CFR is set up to use a better regret minimizer, such as discounted RM or predictive RM+.
- Instead of using the classical self-play scheme described in Fig. 1, people *alternate* the iterates and feedback as described in Fig. 2, where the utility vector $\mathbf{u}_{\mathcal{X}}^{(t)}$ is as defined in (1), whereas

$$\tilde{\mathbf{u}}_{\mathcal{Y}}^{(t)} := -\mathbf{A}^{\top} \mathbf{x}^{(t+1)}.$$

(Note that at the very beginning, \mathbf{x}^1 does not participate in the computation of any utility vector).

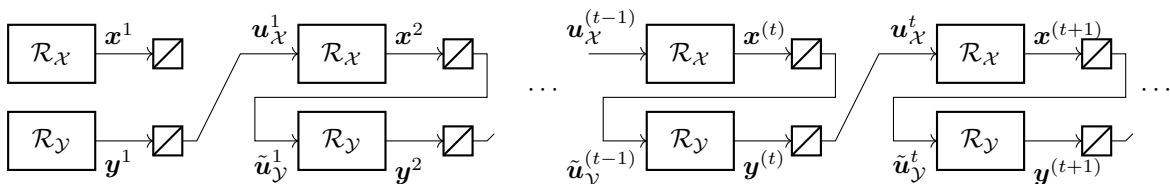


Figure 2: The alternation method for CFR in games. The symbol \boxtimes denotes computation/construction of the utility vector.

- Finally, the output is weighted average of the strategies,

$$\bar{\mathbf{x}}^{(T)} := \frac{1}{\sum_{t=1}^T t^\gamma} \sum_{t=1}^T t^\gamma \mathbf{x}^{(t)}, \quad \bar{\mathbf{y}}^{(T)} := \frac{1}{\sum_{t=1}^T t^\gamma} \sum_{t=1}^T t^\gamma \mathbf{y}^{(t)}$$

is considered instead of the regular averages (2).

5. [5 points] Implement the function `run_dcfp`, which solves games by running CFR with discounted RM as the local regret minimizer, and with alternating iterates. Use hyperparameters $\alpha = 1.5, \beta = 0, \gamma = 2$.
6. [5 points] Implement the function `run_pcfp`, which solves games by running CFR with predictive RM+ as the local regret minimizer, and with alternating iterates. Use $\gamma = 2$.

References

- H. W. Kuhn. A simplified two-person poker. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games*, volume 1 of *Annals of Mathematics Studies*, 24, pages 97–103. Princeton University Press, Princeton, New Jersey, 1950.
- Finnegan Southey, Michael Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and Chris Rayner. Bayes’ bluff: Opponent modelling in poker. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2005.