4180/5180: Reinforcement Learning and Sequential Decision Making (Fall 2024) — Christopher Amato
Northeastern University — Nov. 8th, 2024

Exercise 7: Function Approximation

Please remember the following policies:

- Exercise due at **11:59 PM EST Nov 8, 2024**.
- Submissions should be made electronically on Canvas. Please ensure that your solutions for both the written and programming parts are present. You can upload multiple files in a single submission, or you can zip them into a single file. You can make as many submissions as you wish, but only the latest one will be considered.
- For **Written** questions, solutions may be handwritten or typeset. If you write your answers by hand and submit images/scans of them, please please ensure legibility and order them correctly in a single PDF file.
- The PDF file should also include the figures from the **Plot** questions.
- For both **Plot** and **Code** questions, submit your source code in Jupyter Notebook (.ipynb file) along with reasonable comments of your implementation. Please make sure the code runs correctly.
- You are welcome to discuss these problems with other students in the class, but you must understand and write up the solution and code yourself. Also, you *must* list the names of all those (if any) with whom you discussed your answers at the top of your PDF solutions page.
- Each exercise may be handed in up to two days late (24-hour period), penalized by 10% per day late. Submissions later than two days will not be accepted.
- Contact the teaching staff if there are medical or other extenuating circumstances that we should be aware of.
- **Notations: RL2e is short for the reinforcement learning book 2nd edition. x.x means the Exercise x.x in the book.**

1. **1 point.** (RL2e 10.1) *On-policy Monte-Carlo control with approximation.*
   Written: We have not explicitly considered or given pseudocode for any Monte Carlo methods in this chapter.
   - What would they be like?
   - Why is it reasonable not to give pseudocode for them?
   - How would they perform on the Mountain Car task?

2. **1 point.** (RL2e 10.2) *Semi-gradient expected SARSA and Q-learning.*
   Written:
   (a) Give pseudocode for semi-gradient one-step Expected Sarsa for control.
   (b) What changes to your pseudocode are necessary to derive semi-gradient Q-learning?

3. **4/6 points.** *Four rooms, yet again.*
   Let us once again re-visit our favorite domain, Four Rooms, as implemented in Ex4 Q3 (You can find it in the Notebook). We will first explore function approximation using *state aggregation.* Since this domain has four discrete actions with significantly different effects, we will only aggregate states; different actions will likely have different Q-values in the same state (or set of states).

   (a) **Code:** Implement semi-gradient one-step SARSA using the following state aggregation.

   - Each state is only aggregated to itself. In other words, the aggregated state space is **identical** to the original state space.
   - Since we use SARSA and want to learn Q values, we design the feature for each state(aggregated)-action pair using one-hot encoding. For example, if the size of the aggregate state space is N and the size of the action space is M, then given a state(aggregated)-action pair, you should represent it as a one-hot encoding vector $f = [0, 1, 0, .., 0]_d$, where $d = N \times M$.

- Given the one-hot vector as the feature for each state(aggregated)-action pair, the weights are defined as a vector $w = [w_0, w_1, w_2, ..., w_d]$, where $d = N \times M$.
- Following such design, we can approximate the Q value for one state(aggregated)-action pair as $Q(s, a) = w^T f(s, a) = w_i$, where $f(s, a)$ is the one-hot vector and i is the index of the state(aggregated)-action pair. Obviously, each element in w corresponds to the Q value of one state(aggregated)-action pair. We can call this strategy a *tabular* state aggregation.

Verify that your implementation works by following the design above. Use the default hyperparameters in the Notebook.
Plot: Plot learning curves with confidence bounds, as in past exercises. 100 episodes should be sufficient.
Hint: To check whether your result is correct, you can compare your results against an implementation of tabular SARSA or Q-learning from Ex 5; the results should be similar.

(b) **Code/plot: [5180]** Please implement the following **two** state aggregation strategies, and plot the learning curves.

- Tile-based state aggregation: We define a 2×2 tile and use it to aggregate the state space. For example, the state [0, 0], [1, 0], [0, 1], [1, 1] will be aggregated into one aggregated state. Importantly, the aggregated states are disjoint with each other.
- Room-based state aggregation: We will aggregate the states in one Room into one aggregated state.

Written: Comment on your findings, is aggregating the states helping to boost the learning speed? Shall we always aggregate the states as much as possible to learn better policy?

(c) **[Extra credit.] 1 point.** Implement the semi-gradient one-step Q-learning and resolve (a) and (b) questions above. Comment on whether there is a difference between the results of semi-gradient SARSA and semi-gradient Q-learning?

We will now consider the more general case of linear function approximation.
If necessary, adapt your implementation of semi-gradient one-step SARSA for linear function approximation; this might not be necessary if your implementation is sufficiently general.

(d) One natural idea is to use the (x, y) coordinates of the agent's location as features.
Specifically, use the following three features for state s:

- The state's x coordinate
- The state's y coordinate
- 1 (i.e., the feature value is 1 for any state)

Code/plot: Use these features for linear function approximation, and plot the learning curves.
Written: Why is the constant feature necessary? What do you think happens without it?
Also describe how you incorporated actions into your features.
Written: How do your results with the above features compare with state aggregation?
If there is a significant performance difference, try to come up with explanations for it.

(e) **Code/plot: [5180]** Implement the following **two** features, and plot the learning curves.

- Normalized state coordinates: use the feature $[x/10, y/10, 1]$
- Include distance: use the feature $[d, x, y, 1]$, where d is the shortest distance between the current state to the goal state. (You can find the distance matrix in the Notebook.)

Written: Comment on your findings, including any trends and surprising results.

(f) **[Extra credit.] 1 point.** One of the main benefits of function approximation is that it can handle large state spaces. So far, the Four Rooms domain is still quite small.
Design successively larger versions of Four Rooms, where each grid cell in the original environment can be subdivided into k cells per side (i.e., the number of states expands by a factor of k^2). You can choose whether or not to expand the walls/doorways.
Experiment with the various state aggregation and linear features proposed above (or propose more).
Plot your learning curves, and comment on your findings.

4. **2 points.** *Mountain car.*

   (a) **Code/plot:** Read and understand Example 10.1.
   Implement semi-gradient one-step SARSA for Mountain car, using linear function approximation with the tile-coding features described in the text. Reproduce Figures 10.1 and 10.2.
   Instead of writing your own environment and features, we recommend that you use the implementation of Mountain Car provided by OpenAI Gym, and refer to the footnote on p. 246 for an implementation of tile coding. Make sure you use the discrete-action version (`MountainCar-v0`):
   `https://gymnasium.farama.org/environments/classic_control/mountain_car/`

   Some notes on Mountain Car and reproducing figures:

   - The implementation in OpenAI Gym is close to the book's description, but it has a timeout of 200 steps per episode, so the results you get may be different from that shown in Figures 10.1 and 10.2. This is fine and expected. (If you see footnote 2 on p. 246, you will also see that Figure 10.1 was generated using semi-gradient SARSA(λ) instead of semi-gradient SARSA.)
   - For visualizing the cost-to-go function (Figure 10.1), you can use plotting tools like `imshow` instead of showing a 3-D surface, if you wish.
   - Since episodes time out after 200 steps, for the first subplot of Figure 10.1, just visualize the cost-to-go function at the end of the first episode, instead of after step 428 as shown.
   - If your implementation is too slow, you can omit visualizing the cost-to-go function at episode 9000 (the final subplot in Figure 10.1).
   - When replicating Figure 10.2, it is fine if your vertical axis is a regular linear scale (vs. log scale as shown), since the maximum will be 200 steps per episode. Also, if your implementation is too slow, you can do fewer than 100 trials (e.g., 10 is okay).

   (b) **[Extra credit.] 1 point.** Implement n-step semi-gradient SARSA and reproduce Figures 10.3 and 10.4.

   (c) **[Extra credit.] 1 point.** Read and understand Section 12.7 about SARSA(λ) with function approximation.
   Read and understand Example 12.2, and reproduce Figure 12.10 and the top-left plot of Figure 12.14. (The definition of a replacing trace is given in Equation 12.12.)