CS 4180/5180: Reinforcement Learning and Sequential Decision Making (Fall 2024) — Lawson L.S. Wong
Northeastern University — Due March 15, 2024

Exercise 7: Function Approximation

Please remember the following policies:

- Exercises are due at **11:59 PM** Boston time (ET).
- **Submissions should be made electronically on Canvas.**
  Please ensure that your solutions for both the written and programming parts are present.
  Upload all files in a single submission, keeping the files individual (do not zip them).
  You can make as many submissions as you wish, but only the latest one will be considered, and late days will be computed based on the latest submission.
- Each exercise may be handed in up to two days late (24-hour period), penalized by 5% per day.
  Submissions later than this will not be accepted. There is no limit on the total number of late days used over the course of the semester.
- Written solutions may be handwritten or typeset. For the former, please ensure handwriting is legible. If you write your answers on paper and submit images of them, that is fine, but please put and order them correctly in a single .pdf file. One way to do this is putting them in a Word document and saving as a PDF file.
- Programming solutions should be in Python 3, either as `.py` or `.ipynb` files.
  For the latter (notebook format), please export the output as a PDF file and submit that together. To do so, first export the notebook as an HTML file (File: Save and Export Notebook as: HTML), then open the HTML file in a brower, and finally print it as a PDF file.
- 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, *and* indicate who you discussed with (if any). If you are collaborating with a large language model (LLM), acknowledge this and include your entire interaction with this system. We strongly encourage you to formulate your own answers first before consulting other students / LLMs.
- Typically, each assignment contains questions designated [CS 5180 only.], which are required for CS 5180 students. Students in CS 4180 can complete these questions for extra credit, for the same point values.
  Occasionally, there will also be [Extra credit.] questions, which can be completed by everyone. Point values for these questions depend on the depth of the extra-credit investigation.
- Contact the teaching staff if there are *extenuating* circumstances.

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 points.** *Four rooms, yet again.*
Let us once again re-visit our favorite domain, Four Rooms, as implemented in Ex4 Q4.
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) **Written/code:** Design and implement functions for computing features, approximate Q-values, and gradients for state aggregation. Describe your design.

   (b) **Code:** Implement semi-gradient one-step SARSA (or semi-gradient one-step Q-learning).
   Verify that your implementation works by trying it on *tabular* state aggregation, where each state is actually distinct (i.e., only aggregated with itself). Choose appropriate hyperparameters.
   Plot: Plot learning curves with confidence bounds, as in past exercises. 100 episodes should be sufficient.
   Hint: This is equivalent to the tabular setting, so you can compare your results against an implementation of tabular SARSA or Q-learning from Ex 6; the results should be similar.

   (c) **Code/plot:** Try at least three other choices of state aggregation, and plot the learning curves.
   Are you able to find aggregations that do better than tabular? How about worse than tabular?
   Written: Comment on your findings, including any trends and surprising results.

We will now consider the more general case of linear function approximation.
If necessary, adapt your implementation of semi-gradient one-step SARSA (or Q-learning) for linear function approximation; this might not be necessary if you 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:** Design and implement at least three more features, and plot the learning curves.
   You may use knowledge about the Four Rooms domain, and possibly knowledge about the goal location.
   Written: Comment on your findings, including any trends and surprising results.
   Will your features work if the goal location is not $(10, 10)$?

   (f) **[Extra credit.]** 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. **[CS 5180 only.] 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 Gymnasium, 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 Gymnasium 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).

   The following questions present many opportunities for extra credit. Feel free to do any of these, at any time. You may submit these late without penalty, until the end of the month (3/31). If you do submit these later than your main assignment, please inform the course staff via Piazza.

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

   (c) **[Extra credit.]** 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.)

   (d) **[Extra credit.]** Instead of using linear function approximation using tile-coding features, we can use more sophisticated function approximators such as artificial neural networks (Section 9.7) to *learn* appropriate features. Apply neural-network function approximation to semi-gradient SARSA on Mountain Car.

   (e) **[Extra credit.]** Try to solve the continuous-action version of Mountain Car, `MountainCarContinuous-v0`:
   `https://gymnasium.farama.org/environments/classic_control/mountain_car_continuous`

5. **[CS 5180 only.]** **2 points.** *Residual-gradient TD.*
Written: *Hint*: You may find it helpful to read Section 11.5 on optimizing the Bellman error.
In this question, we consider what changes are necessary to turn *semi*-gradient TD methods into *true*-gradient TD methods. Recall that the semi-gradient methods are derived starting from the mean-squared value error as our overall objective (Equation 9.1):

$$\overline{\mathrm{VE}}(\mathbf{w}) \triangleq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2$$

Since v_π is not actually available to us during learning, we had to substitute appropriate learning targets to perform stochastic gradient descent in order to optimize the objective in practice. By substituting $v_\pi(S) \approx R + \gamma \hat{v}(S', \mathbf{w})$ and ignoring the gradient term on $\hat{v}(S', \mathbf{w})$, we obtained *semi*-gradient TD(0).

(a) Re-derive the learning rule for one-step gradient TD, this time *without* ignoring the gradient term on $\hat{v}(S', \mathbf{w})$. (Do not over-think this, it should be straightforward.)

(b) What objective function does this new learning rule optimize? Note that this will involve an expectation over the next state s'. Is this a good idea? What do you predict will happen?

To obtain a better algorithm, observe that rather than trying to minimize the distance between $\hat{v}(S, \mathbf{w})$ and $R + \gamma \hat{v}(S', \mathbf{w})$, what we really want is to move $\hat{v}(S, \mathbf{w})$ closer to the *expected* value of $R + \gamma \hat{v}(S', \mathbf{w})$, where the expectation is over possible future next states S'.

(c) Consider the following objective function, known as the *mean-squared Bellman error*:

$$\overline{\mathrm{BE}}(\mathbf{w}) \triangleq \sum_{s \in \mathcal{S}} \mu(s) \left[\mathbb{E}_\pi \left[R + \gamma \hat{v}(s', \mathbf{w}) | s \right] - \hat{v}(s, \mathbf{w}) \right]^2$$

Derive a gradient TD-learning rule that optimizes this objective function.

(d) There are some issues with implementing this learning rule in practice.
What are they? Under what circumstances can they be overcome?
Hint: The expectation of a product is not necessarily equal to the product of the expectations.

(e) **Code/plot/written:** Implement the learning rule derived in part (a) and apply it to the Mountain Car environment from Q4. Compare against your semi-gradient SARSA results from Q4.

Similar to the extra-credit questions in Q4, you may submit Q6 by the end of the month (3/31) without penalty. Again, if you submit this later than your main assignment, please inform the course staff via Piazza.

6. **[Extra credit.]** (RL2e 11.3) *Baird's counterexample.*

(a) Apply one-step semi-gradient Q-learning to Baird's counterexample and show empirically that its weights diverge.

(b) Recall the "deadly triad" hypothesis described in Section 11.3 in the textbook.
Verify empirically on Baird's counterexample whether the following statement in the textbook is true:
If any two elements of the deadly triad are present, but not all three, then instability can be avoided.

(c) How stable is Baird's counterexample?
Can you find any small changes to the counterexample that avoids divergence?

(d) Does residual-gradient TD (from Q5) resolve the divergence issue on Baird's counterexample?
Apply your implementation from Q5 to investigate.

(e) Read Ch. 11.7 (and possibly the rest of Ch. 11 for context). Implement TDC and apply them on Baird's counterexample to reproduce Figure 11.5. Note that you may not be able to find the same set of weights (depends on initialization and hyperparameters), but the weights should not diverge and should be approximately correct (similar to Figure 11.5). How does this compare with residual-gradient TD?

(f) Investigate off-policy methods further (including those beyond the textbook) and see if any of them can provide a better solution for Baird's counterexample. Implement, experiment, and discuss your findings.