CS 4180/5180: Reinforcement Learning and Sequential Decision Making (Fall 2021) Robert Platt
Northeastern University Due November 25, 2021

Exercise 7: Function Approximation

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?

The Monte Carlo method is similar to the on-policy Monte-Carlo control, TD learning method as defined by n= T. It is not much difference on the pseudocode. Therefore, it is not reasonable to give pseudocode.

The performance of Monte Carlo method for the task of Mountain Car could be costly as it would not start to learn until the first episode is completed. And also, for each step that does not reach the goal state would receive a negative reward, which make the cost higher and longer to converge for learning compared to TD or SARSA method.

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.

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Choose A' as a function of $\hat{q}(S', \cdot, w)$
$w \leftarrow w + \alpha[R + \gamma \sum_a \pi(a|S')\hat{q}(S', a, w) - \hat{q}(S, A, w)]\nabla\hat{q}(S, A, w)$
$S \leftarrow S'$
$A \leftarrow A'$

(b) What changes to your pseudocode are necessary to derive semi-gradient Q-learning?

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S, and action A
Loop for each step:
  Take action A, observe R, S'
  Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta\delta$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

Choose A as a function of $\hat{q}(S', \cdot, w)$
Take action A, observe R, S'
$\delta \leftarrow R - \bar{R} + \text{argmax}\,\hat{q}(S', a', w) - \hat{q}(S, a, w)$
$\bar{R} \leftarrow \bar{R} + \beta\delta$
$w \leftarrow w + \alpha\delta\nabla(\hat{q}(S, A))$
$S' \leftarrow S$

3. **4 points.** *Four rooms, yet again.*
Let us once again re-visit our favorite domain, Four Rooms.
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.

I design to use different weight vectors for each aggregate states. The weight vector has the same length with aggregations. By using one-hot coding method, the gradient of weight is correlated to the specific feature that needs to be used.

(b) **Code:** Implement semi-gradient one-step SARSA.
Verify that your implementation works by trying it on *tabular* state aggregation, where each state is actually distinct (i.e., only aggregated with itself). Use 100 trials, 100 episodes, epsilon greedy parameter $\varepsilon = 0.1$, discount factor $\gamma = 0.99$, and step size $\alpha = 0.1$.
Plot: Plot learning curves with confidence bounds, as in past exercises.
Hint: This is equivalent to the tabular setting, so you can compare your results against an implementation of tabular SARSA from Ex 5; the results should be similar.

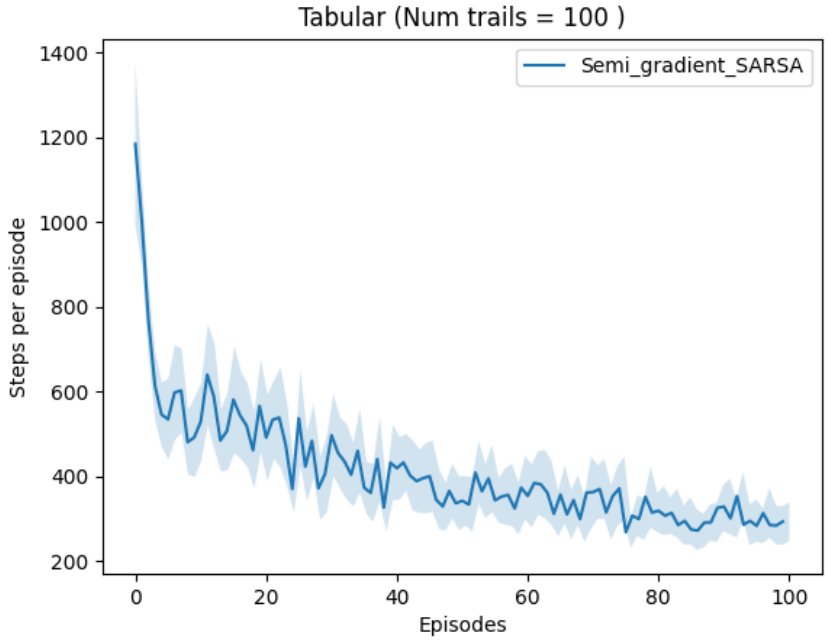

Figure 1. Tabular Equivalence

(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.

Choice 1: row aggregation method by wrapping each row together

Choice 2: room aggregation method by wrapping into four different rooms

Choice 3: 3x3 aggregation method by wrapping the grid world into small groups of size 3 by 3

Choice 4: 2x2 aggregation method by wrapping the grid world into small groups of size 2 by 2

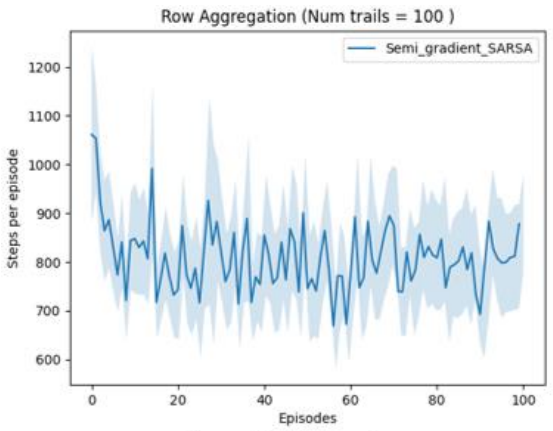

Figure 2. Choice 1

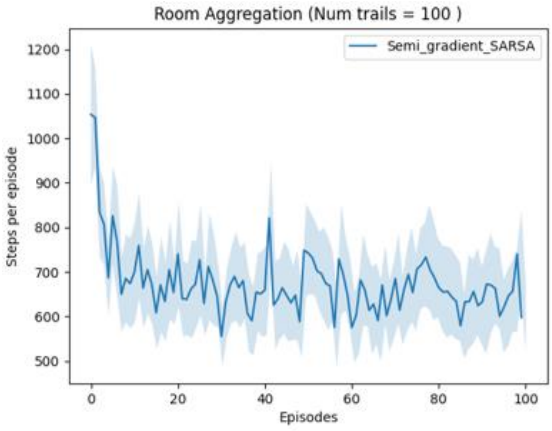

Figure 3. Choice 2

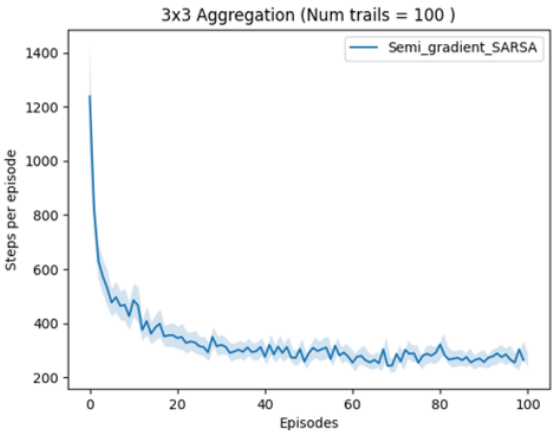

Figure 4. Choice 3

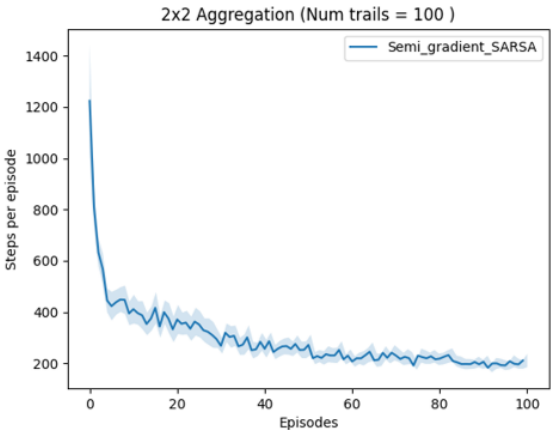

Figure 5. Choice 4

The smaller wrapping fromat leads to the graph more similar to the tabular one. Larger aggregation decrease earlier than smaller ones. The result is not surprising since less quantities will update more efficient than more quanties(tabular one). Therefore, the tabular converges faster than state aggregation wrapping. But the running speed is slower.

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.

The constant is necessary as it work as the bias term in the normal regression method. Without a bias term, the function would always across (0, 0) no matter what update that made. It would cause large error.

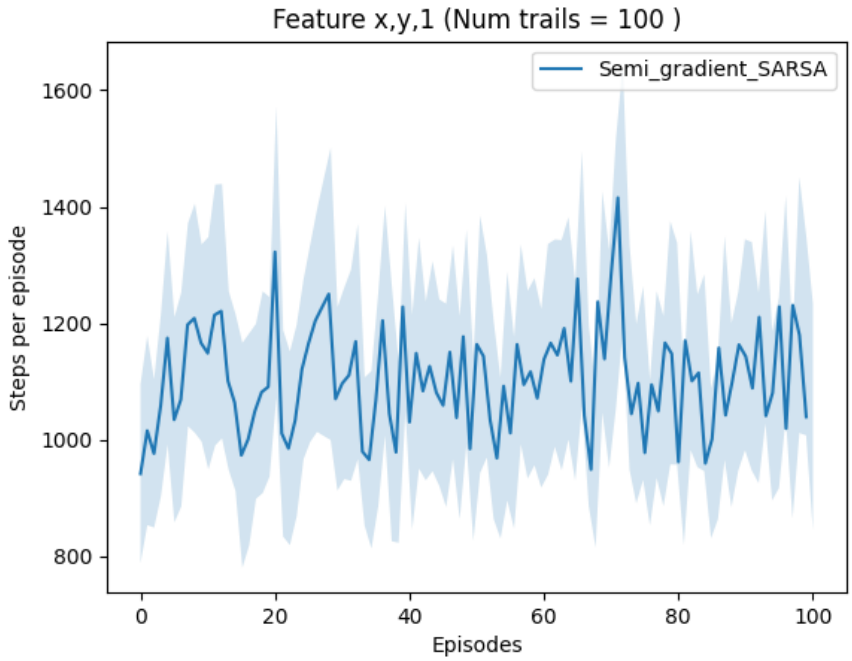

Figure 6. Feature x, y, 1

Compared to the result of state aggregation, the feature 1, x, y is not as good as the state aggregation one. It is close to a line feature and the learning result is bad and a huge difference. It is because that the number of features is not enough to estimate.

(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 $(10, 10)$?

Choice e_1: room features

Choice e_2: room number features

Choice e_3: goal distance features (1, x, y, xy)

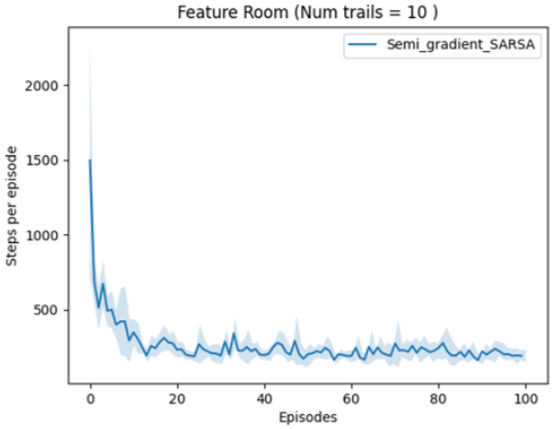

Figure 7. Choice e_1

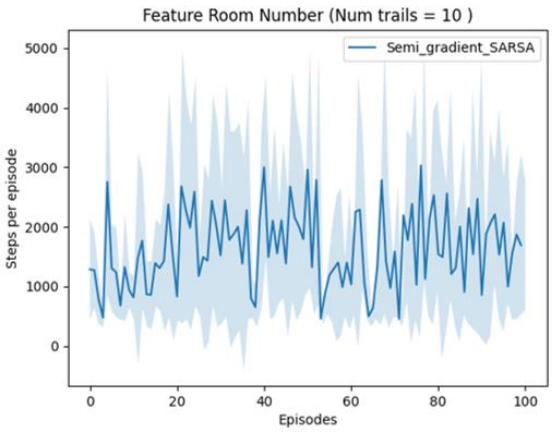

Figure 8. Choice e_2

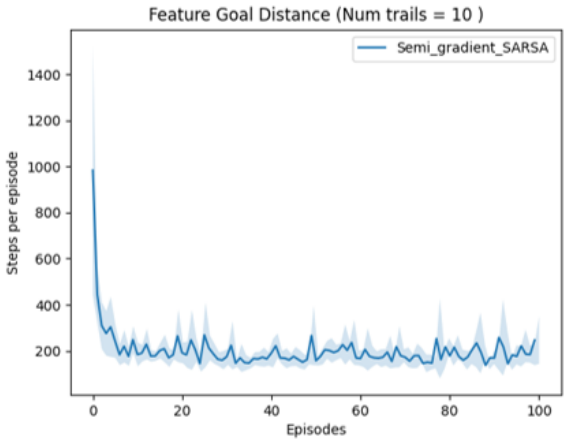

Figure 9. Choice e_3

Both the feature of room and goal distance is showed with a sharp decrease in the steps, but the feature of room number is closed to a linear format. It is surprising to find the feature with room number is not perform as expected.

The feature will still be working if the goal location is not (10, 10) but the performance might be varied based on the different goal states.

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://gym.openai.com/envs/MountainCar-v0`

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.
- When replicating Figure 10.2, use a regular linear scale rather than a log scale as shown, since the maximum will be 200 steps per episode.

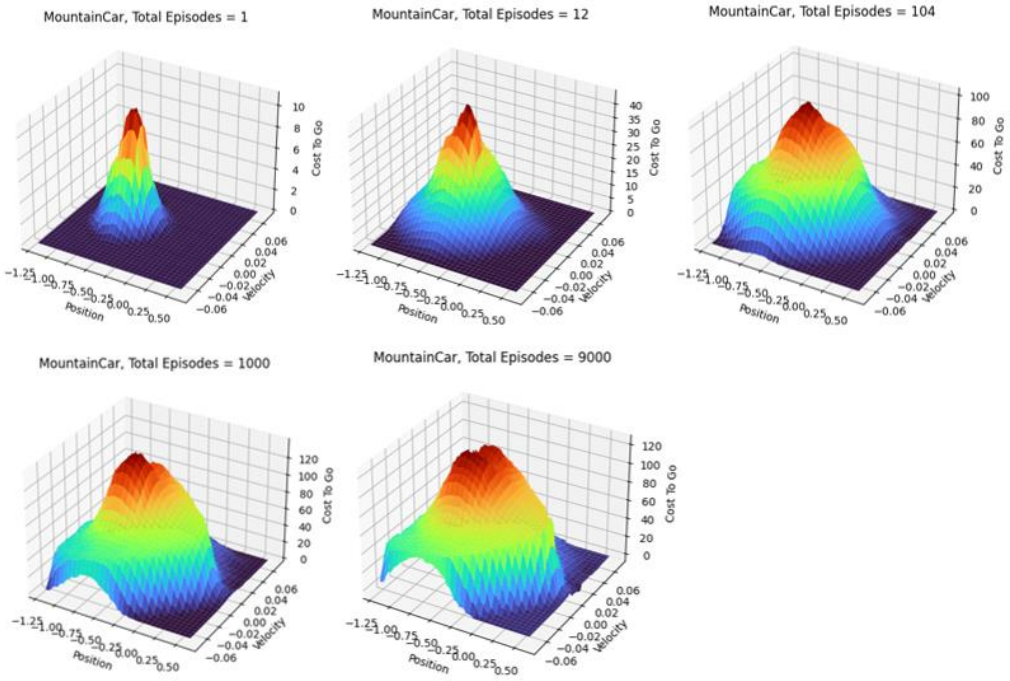

Figure 10. Mountain Car with Cost to Go Function

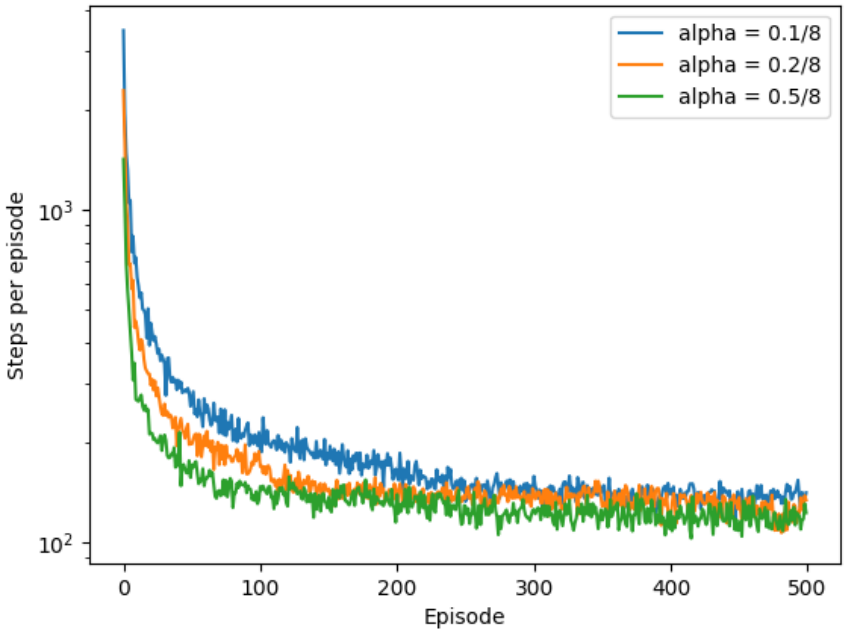

Figure 11. Mountain Car learning curves (log)

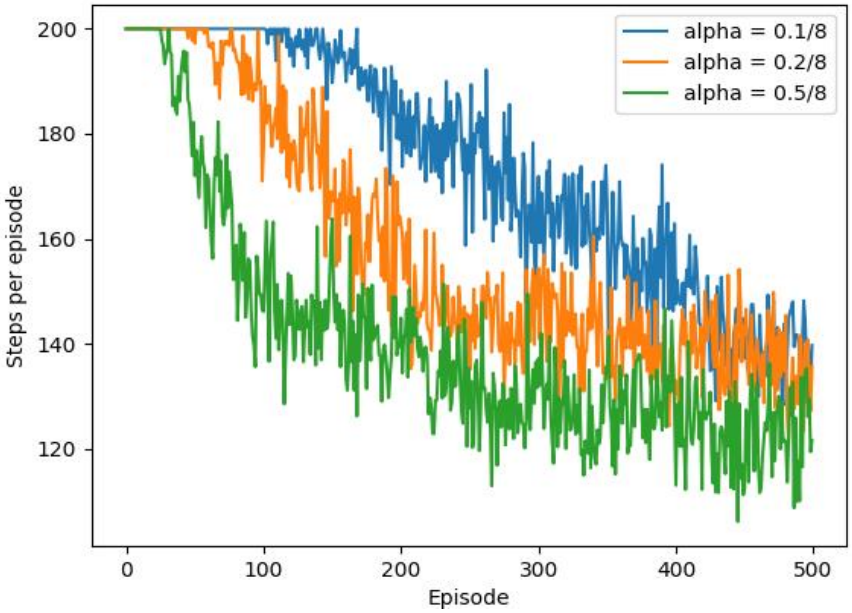

Figure 12. Mountain Car learning curves

The following questions present many opportunities for extra credit.

(b) **[Extra credit 0.5 points.]** Implement n-step semi-gradient SARSA and reproduce Figures 10.3 and 10.4.

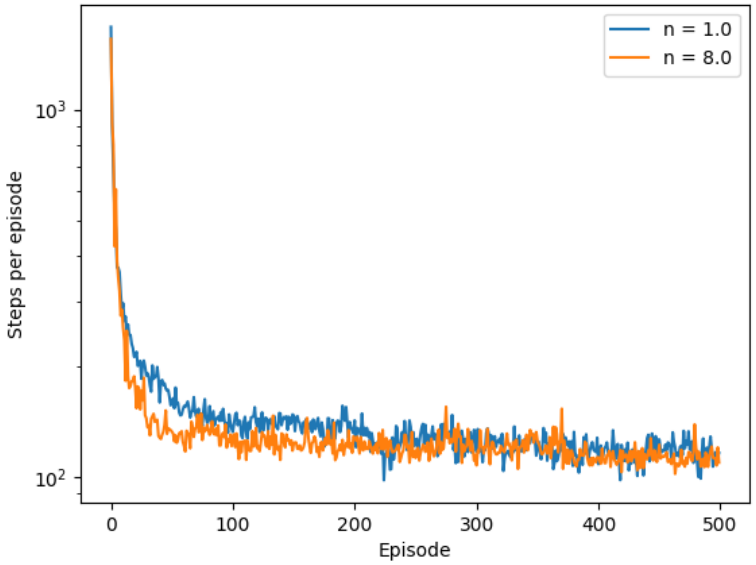

Figure 13. Mountain Car Performance of one-step vs 8-step

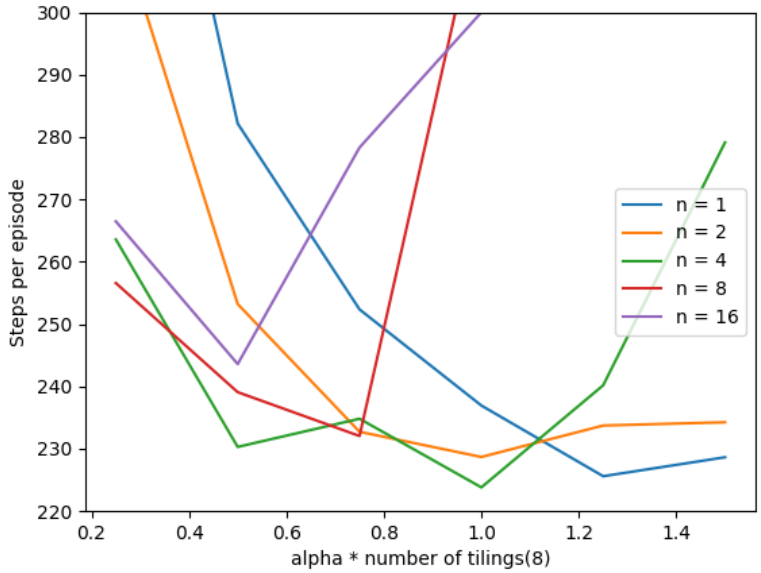

Figure 14. Mountain Car Performance Effect of α and n

5. **[CS 5180 only.] 1 point.** *Residual-gradient TD.*
Written: *Hint*: For this question, you may find it helpful to read Section 11.5 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):

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

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.)

$$v_\pi(s) = R + \gamma \hat{v}(S', w)$$

$$\begin{aligned}\overline{VE(w)} &\triangleq \sum_{s \epsilon S} \mu(s)[v_\pi(s) - \hat{v}(s, w)\,]^2 \\ &\triangleq \sum_{s \epsilon S} \mu(s)[R + \gamma \hat{v}(S', w) - \hat{v}(s, w)\,]^2 \\ &= E[R + \gamma \hat{v}(S', w) - \hat{v}(S', w)]^2\end{aligned}$$

$$\begin{aligned}w_{t+1} &= w_t - \frac{1}{2}\alpha \nabla \overline{VE(w)} \\ &= w_t - \frac{1}{2}\alpha \nabla [E(R + \gamma \hat{v}(S', w) - \hat{v}(S', w))]^2 \\ &= w_t + \alpha E[(R + \gamma \hat{v}(S', w) - \hat{v}(S', w))(\nabla \hat{v}(S', w_t) - \gamma \nabla \hat{v}(S', w_t))]\end{aligned}$$

(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?

By replacing $v_\pi(s)$, the TD error will be minimized from the objective function as followed:

$$\overline{VE(w)} = E[R + \gamma \hat{v}(S', w) - \hat{v}(S', w)]^2$$

It is not a good idea to involve expectation over S' as the loss function depends on next state. This will lead to a bias that changes the converge state and value.

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*:

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

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

$$\begin{aligned}
\overline{BE(w)} &\triangleq \sum_{s \epsilon S} \mu(s)[E_\pi[R + \gamma\hat{v}(S', w)|s] - \hat{v}(s, w)\,]^2 \\
&= E[(E_\pi\left[R + \gamma\hat{v}(S', w)\middle|s - \hat{v}(s, w)\right)^2\Big] \\
&= 2E\Big[\big(E_\pi\big[R + \gamma\hat{v}(S', w) - \hat{v}(s, w)\big)\big(E_\pi\big[\gamma\nabla\hat{v}(S', w) - \nabla\hat{v}(s, w)\big)\Big]
\end{aligned}$$

$$\begin{aligned}
w_{t+1} &= w_t - \frac{1}{2}\alpha\nabla\overline{BE(w)} \\
&= w_t - \alpha E[E_\pi[(R + \gamma\hat{v}(S', w) - \hat{v}(s, w)](\nabla\hat{v}(s, w) - \nabla E_\pi[R + \gamma\hat{v}(S', w)]) \\
&= w_t + \alpha E[E_\pi[(R + \gamma\hat{v}(S', w) - \hat{v}(S', w)](\nabla\hat{v}(s, w) - \gamma E_\pi[\nabla\hat{v}(S', w)])
\end{aligned}$$

(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.

In the approaching method above, the problem is that the formulation involves production of two expectations. Both independent samples of the next state are required to get unbiased sample. But for it is not promised that two examples are equal except they are independent to each other. In the real world with some normal interaction circumstances, only one expectation is needed. Therefore, this algorithm is biased and will be converged on a wrong value. For circumstances like deterministic environment, where the transition to next state is determined, the two samples will be the same and therefore the algorithms should work properly. Or in simulated environment that independent sample obtain to work for different expectations, then the algorithms could converge to the minimize value.