Amina Tabassum

NUID: 002190127

Exercise - 7

Question No: 1

The Monte Carlo method will collect reward at each time step but update value only at the end of episode (kind of like n-step SARSA where n is the number of steps in each episode). So, it will collect reward at each time step, calculate return at each time step but will update values at the end of episode.

I can think of two reasons:

1) MC method updates values at the end of episode which means it is more time consuming especially when we don't know how long an episode will be.
2) MC methods have large variance. Improper features or inefficient state aggregation will result in larger variance.

If we use Monte Carlo to train mount car agent, we will have to wait for completion of episode. So, agents do not learn during transitions. Plus, there will be variance during this learning process as MC methods have high variance and low bias.

Question: 02

a) Pseudo Code for Semi-gradient one step Expected SARSA control

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)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma \sum_a \pi(a|S')\hat{q}(S', a, w) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

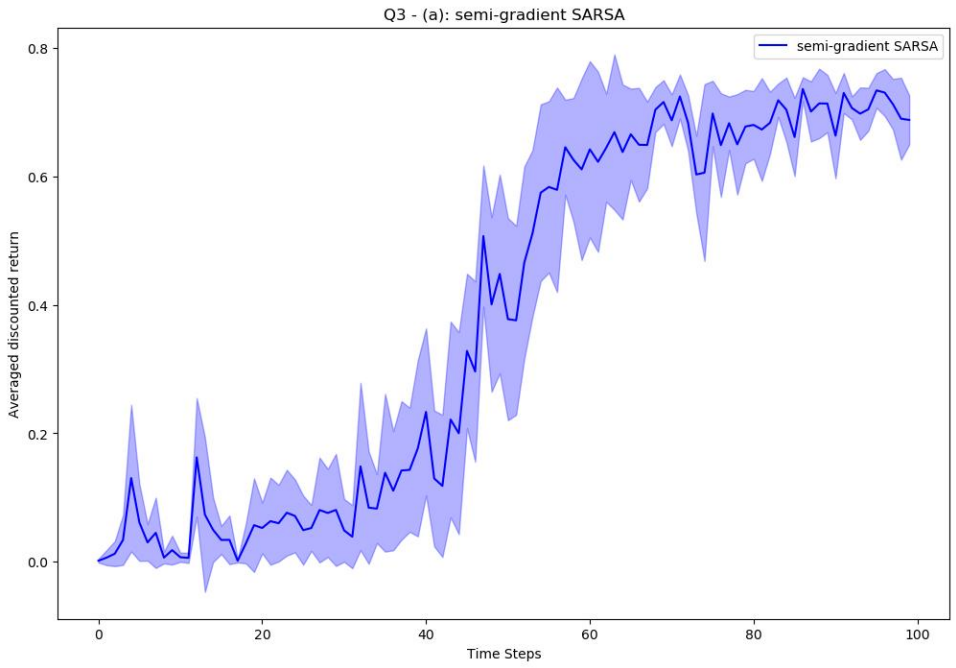
b) Pseudo Code for Semi-gradient Q-learning

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., $\mathbf{w} = \mathbf{0}$)

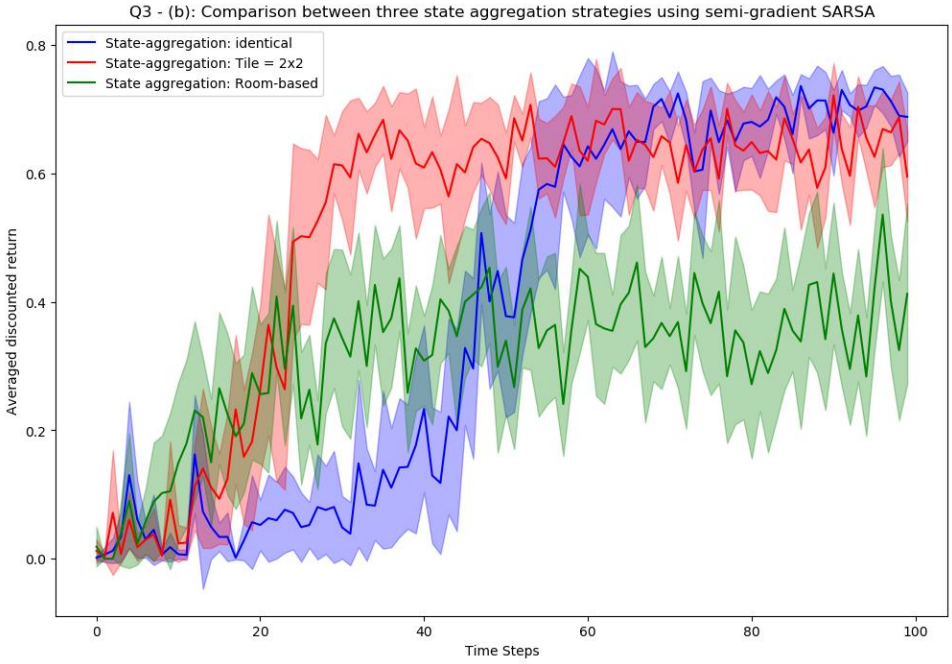
Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$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

Question 03

a)



b)



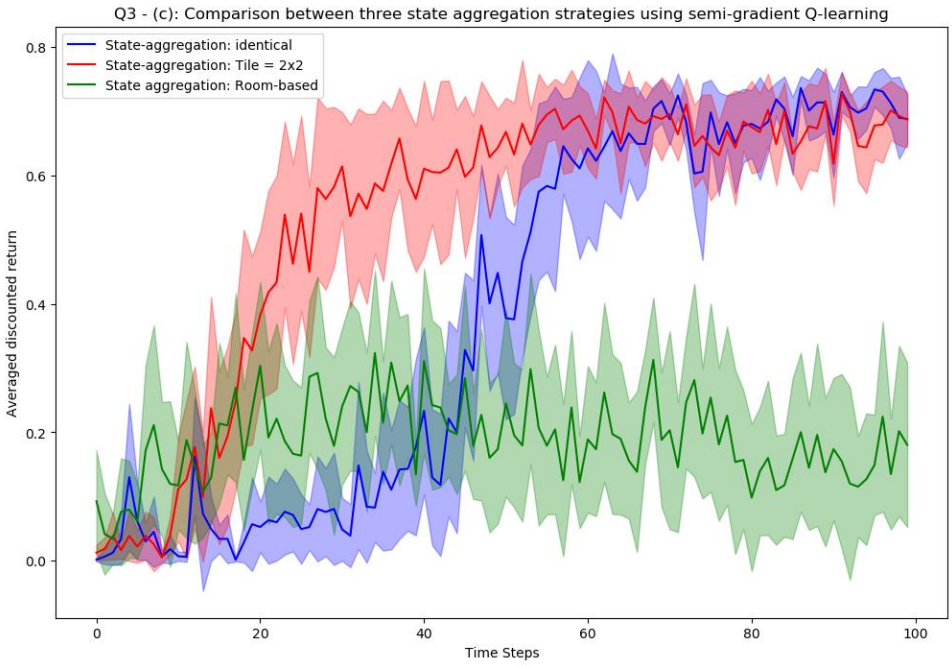
Discussion

The above plot shows three different state aggregation strategies. Blue line indicates state aggregation strategy when each state is aggregated to itself (Total number of states=104) .Red line indicates tile based state aggregation where all four states are aggregated to one state (Total number of states = 35) and green line indicates room based state aggregation where I considered each room as one state and every door as one state (Total number of states = 8). As number of states decreases I.e. state aggregation, agent learns more quickly since it does not have to explore a lot. Whereas, with larger number of states, agent learns to finish episode more quickly. So, with lower number of states, agent learns relatively fast and settles to a value more quickly. Considering the case of room based aggregation, the agent learnt faster, updated values of each state action pair and hence settled to value giving average discounted return of 0.2. It did not explore.

Interesting case is to compare identical and tile based state aggregation. Here, tile based state aggregation agent performed well. It explored it's surrounding states , learnt and settled to value giving average return comparable to identical state aggregation. So, tile based state aggregation gave us similar performance with 35 number of states as compared to identical states aggregation with 104 states. So, lower memory consumption and less computation time.

Conclusion: Appropriated state aggregation does boost the training process! So, we should perform optimal state aggregation (one to ensure both exploration and learning) to learn better policy.

c)



Discussion

There is not huge difference between results from Q-learning and one step SARSA except that variance is slightly higher for tile based state aggregation when it has explored all states and about to settle for certain value. Q-learning selects greedy action during update step so, (for sake of comparison ignoring the variance parameter) , it initially selects the best action and hence value rises but since it is still exploring so, this peak reduces slightly. Other than this, Q-learning learning curve is smoother than semi-gradient SARSA.

d)

This constant term in features is constant term in regression model. It acts as bias term. If we remove it, the agent will always cross (0,0) no matter what update it has taken hence, error will be large.Hence, agent will not learn quite effectively as shown in figure below:

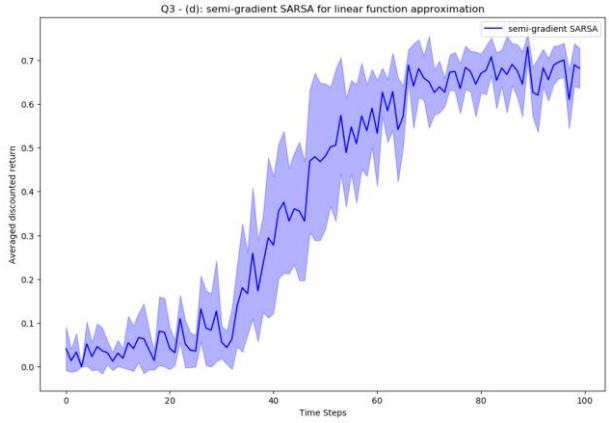


Figure: Linear Function approximation with constant term removed

Another interpretation of this constant can be that it is used as a reference by agent since for max-min scaling, the agent needs to understand what does value between and 0 and 1 mean. For instance, an increment of 5000 is greater than increment of 500. But, increment of 5000 for value capped at 50000 is not very effective as compared to increment of 500 for value capped at 500. So, this constant term provides reference. It compares how much x or y has changed with respect to constant 1.

In order to incorporate actions into feature vector, I created feature vector as :

Feature vector =[x y 1, up,down,left,right]

Where up,down, left and right is selected by hot encoding I.e, it will be one for selected action.

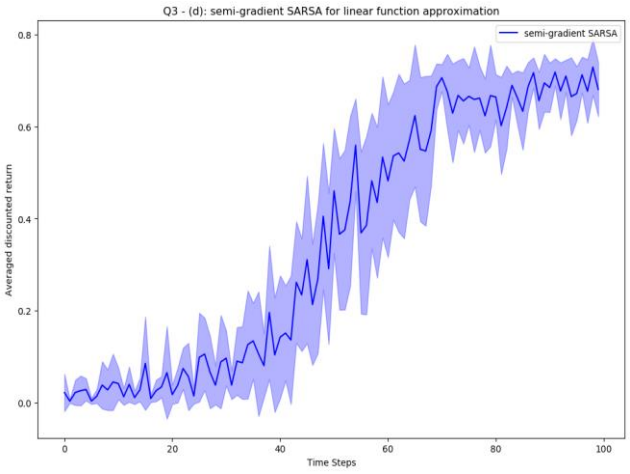
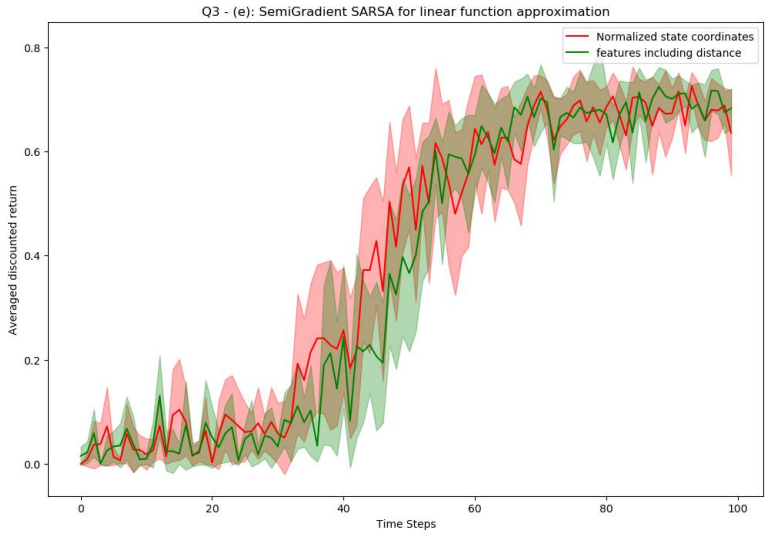


Figure: Linear Function Approximation

The results are not as good as state aggregation because we do not have as many features as previously. Previously, we had features for every state action pair but here we have 7 features for 104 x 4 state action pairs. Hence, agents do not learn quite effectively and quickly. I t is almost always exploring.

e)



Discussion

It looks like that agent settles to certain value in given number of time steps for both cases. Initially, the agent explores both cases. However, agents learn relatively faster for features using normalized state coordinates as compared to features using both distance and state coordinates. It has higher variance as well. A possible reason I can think of is that using x and y as features, agent wants to give higher weightage to actions avoiding dead end/wall and hence explore, learn and settle faster with lower variance.

f)

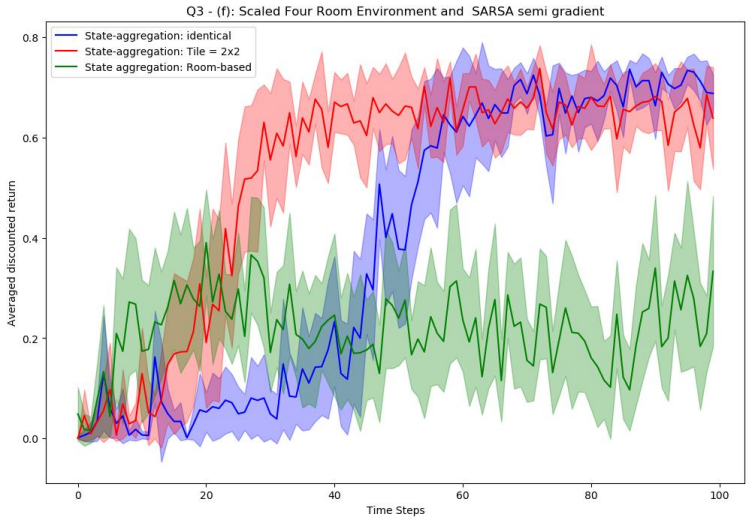


Figure: Scaled Room environment and different state aggregation strategies

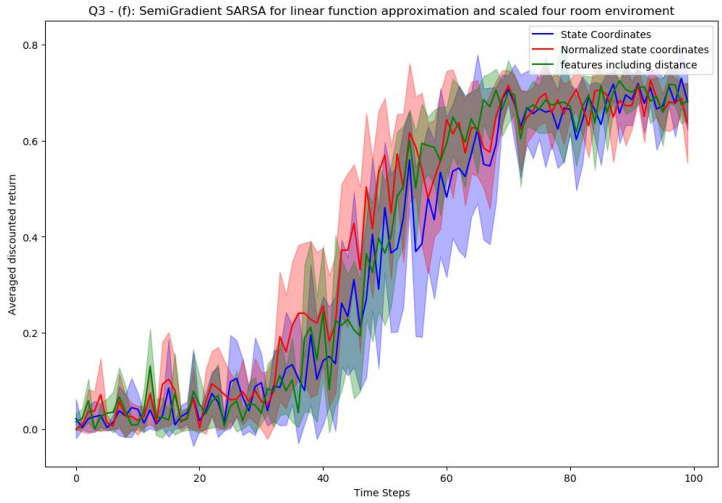


Figure: Scaled room environment and linear function approximation

For this part, I wrote a generalized extended grid cell function which can scale basic four room environment by k. I scaled it by 2 so, it produced 55 by 55 room space. The results for both state aggregation strategies and linear function approximation are attached herewith. Scaled environment also performs like basic environment except here agents take so long to learn. It has to explore bigger area and hence it takes a very long time to reach goal position. In our case, I used k=2 , as a result I set maximum time steps to 100000 because for few episodes it did not even reach goal position in 50,000 time steps. Then, for sake of comparison plot with simple environment, I scaled down these steps to scale of range of 100. Computational efficiency previously was O(k^2) which now has increased to O(k^2) * O(k^2) .