AI 533 **Intelligent Agents and Decision Making** Winter 2025

OREGON STATE UNIVERSITY

School of Electrical Engineering and Computer Science

Name: Hyuntaek Oh

Email: ohhyun@oregonstate.edu

Total points: 100 Mini Project 2 Due date: March 10, 2025

Instructions: This project consists of two parts. Collaboration is not allowed on any part of this project. You are welcome to brainstorm coding practices (including data structures to use) or clarify your understanding of the question with your peers but you cannot code together or copy/re-use solutions. Document who you worked with for this project and cite websites from which you utilized any code (e.g., Stack Overflow, Stack Exchange, ChatGPT) in your code implementation. **You are not allowed to use AI tools to write the code for you.** You must submit a project report (.pdf file) and the code.

Problem Setup We will build on the gridworld used in Mini-project 1. Consider the gridworld in Figure 1, with two states covered in water and two states with wildfire. Each state is denoted as $< x, y, water, fire >$. "Water" and "fire" are binary values, with 0 denoting the absence of water/fire and 1 denoting the presence of water/fire at location (x, y). The start state is denoted as $< 0, 0, 0, 0 >$ and the goal state is denoted by $< 3, 3, 0, 0 >$. The agent can move in all four directions. The agent succeeds with probability 0.8 and may slide to the neighboring cells with probability 0.1. Illustration of the transition probability for actions 'up' and 'right' are shown in Figure 1. If a move is invalid (such as moving into a wall), the agent will remain in that state with the corresponding probability. For example, when trying to move up in top-left cell (start state) of the grid, the agent will remain in that cell with probability 0.9 or move right with probability 0.1.

The agent receives a reward of $+100$ when it reaches the goal state. The agent receives a reward of -5 in the water states, -10 in wildfire states, and a reward of -1 in all other states. The process **terminates** when the agent reaches the goal state. The agent's objective is to maximize the expected reward it can obtain.

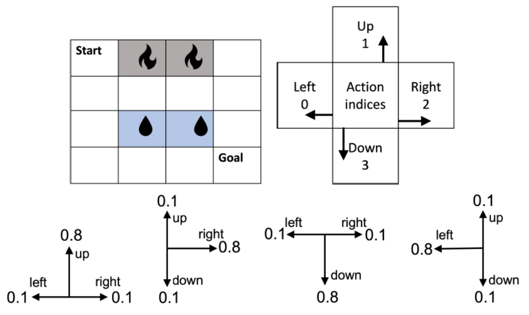


Figure 1: MDP

Figure 1: MDP

Part A: TD Methods (60 points)

1. **(20 points)** Implement tabular SARSA for this problem with $\gamma = 0.95$, using $\epsilon-$greedy action selection. You may optimize the hyperparameters (α, ϵ) either manually or using automated search (e.g. grid search or random search in the space of hyperparameters). Report the hyperparameters you use. Include learning curve over 100 episodes in the .pdf file. Average results over 100 trials, and include standard deviation error bars. You may use an initial policy of your choice but clearly state the initial policy in the document.

Tabular SARSA was implemented with $\gamma = 0.95$, using $\epsilon-$greedy action selection. The update rule of the SARSA algorithm is:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$$

In the SARSA code, the hyperparameters such as (α, ϵ) were optimized by using random search method, which is an optimization algorithm used for finding the optimal or near-optimal solution in a given search space. The random search was conducted with 20 times of random sampling and testing for each sample in 20 trials (=2,000 episodes). The range of the hyperparameter α was between 0.01 and 0.5, and the range of the hyperparameter ϵ was between 0.1 and 0.9. The optimized hyperparameters α and ϵ used in this experiment were 0.4689 and 0.4589 respectively.

For initial policy, a 4 by 4 Q-value table with initialization to zero for all state-action pairs was used at the beginning of the program, and then $\epsilon-$greedy action selection without decaying was applied into the Q-value table. At the first step, all Q-values have same value of zero, the agent selected action randomly. After a few steps, the Q-values updated many times, meaning that future value calculation is proper for $\epsilon-$greedy policy.

According to the lecture note, $\epsilon-$greedy action selection is based on the value of ϵ and the total number of actions. In the experiment, the value of ϵ and the number of actions are 0.4589 and 4 respectively. This can be calculated with the equation:

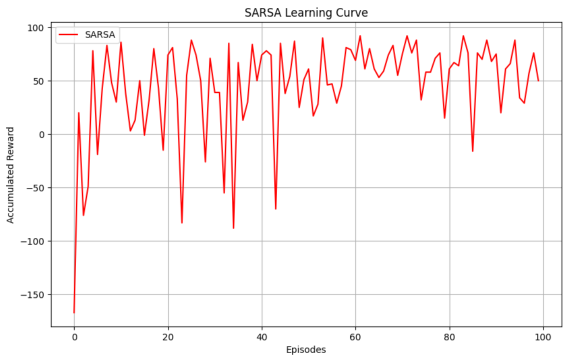
$$\pi(s,a) = \left\{ \begin{matrix} 1 - \epsilon + \dfrac{\epsilon}{|A|} \; if\, a = A^* \\ \dfrac{\epsilon}{|A|} \; if\, a \neq A^* \end{matrix} \right\}$$

$$, \text{where } A^* \in arg\max_a Q(s,a)$$

Based on the equation above, the probability of a greedy action is about 0.65585, and the probability of a non-greedy action(prefer to exploration) is about 0.114725. There are four actions (left, up, right, down), meaning that if left is greedy action, the probabilities would be $[0.65585,\ 0.114725,\ 0.114725,\ 0.114725]$. The total sum of these probabilities should be 1. Thus, next action would be selected randomly by the probabilities above.

The results of the tabular SARSA algorithm is in Figure 2. The experiment was conducted by running the SARSA algorithm 100 episodes for plotting its learning curve. As can be seen in the SARSA Learning Curve, the first 10 episodes shows negative episodic rewards such as -80 and -170, and then it gradually converges nearly 50 even though there are some fluctuations derived from the probabilities of action selection and transitional function, which can cause inaccurate value update.

In the case of the Error bar, on the other hand, the experiment was carried out 100 trials (=10,000 episodes) to identify how the algorithm converges. For each trial, Q-value table initialized to init Q-value table with zeros. The graph (right) in Figure 2 displays the convergence of the tabular SARSA algorithm in a intuitive way, indicating that the converged value was about 75.



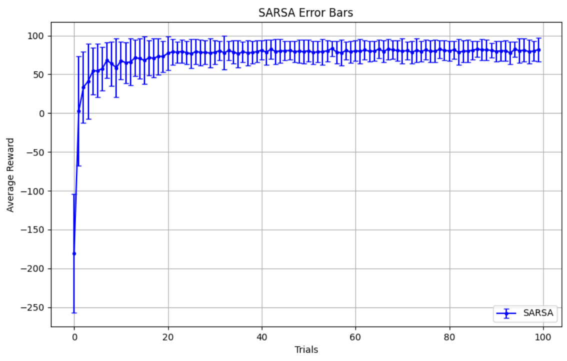


Figure 2: SARSA Learning Curve(left) and Error bar(right)

2. **(20 points)** Implement tabular Q-learning for this problem with $\gamma = 0.95$, using $\epsilon-$greedy action selection. You may optimize the hyperparameters (α, ϵ) either manually or using automated search (e.g. grid search or random search in the space of hyperparameters). Report the hyperparameters you use. Include learning curve over 100 episodes in the .pdf file. Average results over 100 trials, and include standard deviation error bars. You may use an initial policy of your choice but clearly state the initial policy in the document.

Tabular Q-learning algorithm was implemented with $\gamma = 0.95$, using $\epsilon-$greedy action selection. The update rule of the Q-learning algorithm is:

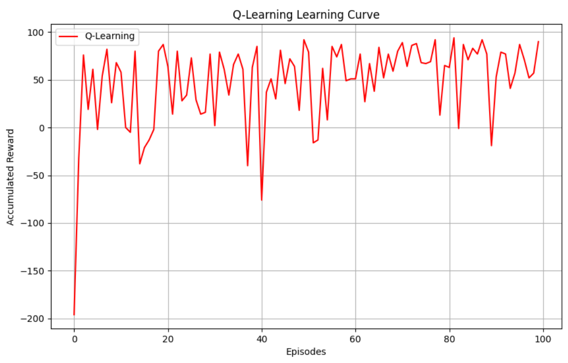
$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$$

Likely, the hyperparameters α and ϵ in the Q-learning code were optimized by random search method. In the random search method, the search space of Q-learning algorithm was as same as one of the SARSA algorithm, (0.01, 0.5) for α and (0.1, 0.9) for ϵ. The number of testing parameters was also 20 trials (=2,000 episodes). The optimized parameters α and ϵ were 0.4908 and 0.74922 respectively.

For initial policy, a 4 by 4 Q-value table with initialization to zero for all state-action pairs was used at the beginning of the program, and then $\epsilon-$greedy action selection without decaying was applied into the Q-value table.

$\epsilon-$greedy policy with non-decaying was used, and the equation of $\epsilon-$greedy action selection. A 4 by 4 Q-value table initialized to zero for all state-action pairs was used at the first step of the program. The process of applying $\epsilon-$greedy action selection into the Q-value table worked in a identical way the SARSA algorithm used. Based on the equation of $\epsilon-$greedy action selection, the probability of a greedy action in Q-learning was 0.438085, and the probability of a non-greedy action in Q-learning was 0.187305.

As shown in Figure 3 (left), the experiment was implemented by running the Q-learning algorithm with 100 episodes for plotting its learning curve. It implies that the learning curve of Q-learning converges near 50-60 even though the agent earned nearly -200 negative reward at the first episode. For the Error bar in Figure 3 (right), the test was taken 100 trials (=10,000 episodes), and it shows the convergence of Q-learning algorithm. It grows drastically at first and keep steady and stable state, about 75, without significant fluctuation.



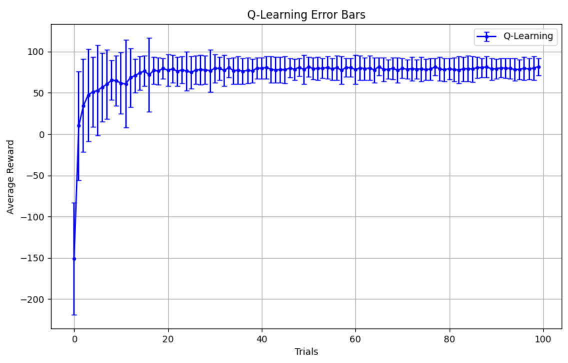


Figure 3: Q-Learning algorithm Learning Curve(left) and Error bar(right)

3. **(20 points)** Implement tabular SARSA(λ) for this problem with $\gamma = 0.95$, using $\epsilon-$greedy action selection. Use the backward algorithm (with eligibility traces). You may optimize the hyperparameters (α, ϵ) either manually or using automated search (e.g. grid search or random search in the space of hyperparameters). Report the hyperparameters you use. Include learning curve over 100 episodes in the .pdf file. Average results over 100 trials, and include standard deviation error bars. You may use an initial policy of your choice but clearly state the initial policy in the document.

Tabular SARSA(λ) algorithm was implemented with $\gamma = 0.95$, using $\epsilon-$greedy action selection and the backward algorithm (with eligibility trace). The update rule of the SARSA(λ) algorithm is:

$$\text{TD error: } \delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$$

$$\text{Eligibility trace: } e(s, a) \leftarrow e(s, a) + 1$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$$

$$e(s, a) \leftarrow \gamma\lambda e(s, a)$$

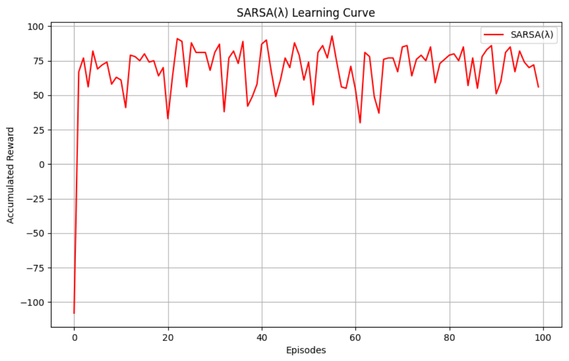
The hyperparameters α and ϵ in the SARSA(λ) code were optimized by random search method, which is used in SARSA and Q-learning algorithms. In the random search method, the search space of SARSA(λ) algorithm was as same as one of the other algorithms used, (0.01, 0.5) for α and (0.1, 0.9) for ϵ. The number of testing parameters was also 20 trials (=2,000 episodes). The optimized parameters α and ϵ were 0.4306 and 0.2573 respectively.

For initial policy, a 4 by 4 Q-value table with initialization to zero for all state-action pairs was used at the beginning of the program, and then $\epsilon-$greedy action selection without decaying was applied into the Q-value table.

$\epsilon-$greedy policy without decaying was used, and the equation of $\epsilon-$greedy action selection. A 4 by 4 Q-value table initialized to zero for all state-action pairs was used at the first step of the program. The process of applying $\epsilon-$greedy action selection into the Q-value table worked in a similar way other previous algorithms used. Based on the equation of $\epsilon-$greedy action selection, the probability of a greedy action in Tabular SARSA(λ) was 0.807125, and the probability of a non-greedy action in SARSA(λ) was 0.064425.

Unlike other algorithms, λ was used to determine how much past rewards and experiences influence the current update. The fixed value of λ in the SARSA(λ) code was 0.8, which means that some past actions influence learning, but older actions decay, reducing variance. Moreover, eligibility trace was utilized to count *the visits of each state-action pair (s,a), maintaining a trace e(s,a). e(s,a) represents how eligible it is for updates.*

As can be seen in Figure 4 (left), the experiment was implemented by running the SARSA(λ) algorithm with 100 episodes for plotting its learning curve. It indicates that the learning curve of SARSA(λ) converges near 70 even though the agent earned nearly -120 reward at the first episode. For the Error bar in Figure 4 (right), the test was taken 100 trials (=10,000 episodes), and it shows the convergence of SARSA(λ) algorithm. It also grows drastically at first and keep steady, stable state, about 75, as like other previous algorithms.



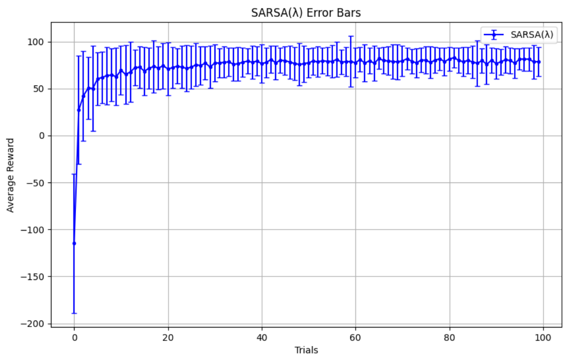


Figure 4: SARSA(λ) Learning Curve(left) and Error bar(right)

Part B: Actor-Critic (40 points)

1. **(25 points)** Implement actor-critic algorithm for this problem with $\gamma = 0.95$ and a *function approximator* of *your choice*. clearly state the initial policy in the submission file. Report the best hyperparameter values for your setting and how you identified them. Include plots of the learning curve across 100 episodes, averaged over 100 trials, along with the standard deviation in the .pdf file.

TD Actor-Critic algorithm was implemented with $\gamma = 0.95$ and a linear functional approximator. The update rule of TD Actor-Critic algorithm is:

$$\textit{TD error: } \delta_t = r_t + \hat{V}_w(s_t + 1) - \hat{V}_w(s_t)$$

$$\textit{Critic parameter: } w \leftarrow w + \beta \cdot \delta_t \cdot \nabla_w \hat{V}_w(s_t)$$

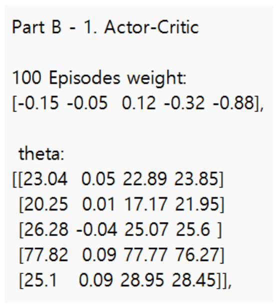
$$\textit{Actor parameter: } \theta \leftarrow \theta + \alpha \cdot \delta_t \cdot \nabla_\theta log\pi_\theta(s_t, a_t)$$

For initial policy, w and θ at the "set up the environment section" in the code are defined as:

$$w = np.random.uniform(-0.01, 0.01, feature_dim)$$

$$theta = np.random.uniform(-0.01, 0.01, (feature\_dim, number\ of\ actions)),$$

where the feature dimension is 5, which is the size of the feature vector (it will be explained at the linear approximator part). w and θ consist of random small float values individually. These values will be used to calculate initial action selection. The Figure 5 shows an example of w and θ, which is final result.



```
Part B - 1. Actor-Critic

100 Episodes weight:
[-0.15 -0.05  0.12 -0.32 -0.88],

 theta:
[[23.04  0.05 22.89 23.85]
 [20.25  0.01 17.17 21.95]
 [26.28 -0.04 25.07 25.6 ]
 [77.82  0.09 77.77 76.27]
 [25.1   0.09 28.95 28.45]],
```

Figure 5: The example of w and theta

Softmax action selection with temperature tau(τ) was implemented to choose an action being taken. By transforming the given state into feature vector, the feature vector was multiplying with Actor parameter θ to identify preference(=logits). Next, this preference was controlled by temperature tau(τ) for balancing exploration and exploitation. The value of tau was affected by the number of episode as exponential decaying:

$$\tau = max(0.1,\ 1.0 * np.exp(-0.03 * episode))$$

At first step, the value of tau started at 1.0, meaning that the policy was to choose actions more randomly. Over time step, it decayed until becoming 0.1, which is the lowest boundary to avoid Nan values and fluctuations.

The hyperparameters α and β are optimized by using random search method as well. The search space of hyperparameters α is (0.001, 0.07), and the value of β is based on the α's search space. It has the 2-5 times of α value for tuning:

$$\beta \simeq [2\alpha,\ 5\alpha]$$

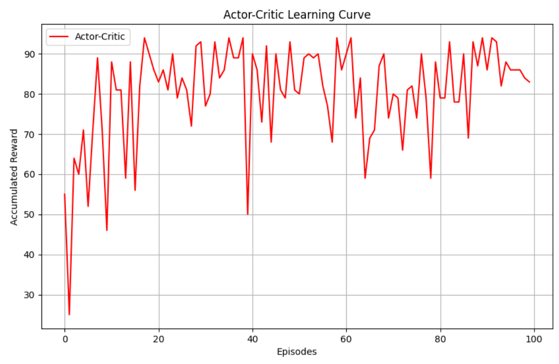
For tuning hyperparameters, 20 times of sampling and 100 trials (=10,000 episodes) were carried out. The optimized hyperparameters α and β was 0.0548 and 0.1643 respectively.

In the Actor-Critic code, a linear funtional approximator was used to reduce computational cost. For the linear approximator, the given state was converted into a feature vector. There are 5 features in feature vector, including row, column, distance between current position and water, distance between current position and fire, and distance between current position and goal.

Then, the feature vector was used to calculate the value function of current or next state by multiplying with the Critic parameter weight w. The temporal difference value was generated by calculating these value functions, gamma, and the given reward. When Critic parameter w is updated, moving average method was utilized to reduce short-term fluctuations and highlighting longer-term trends.

Actor parameter θ was updated by the result of policy gradient method in the lecture note. In the Actor-Critic code, the output of policy gradient was clipped since there are some extremely large or small values during training.

As shown in Figure 6 (left), the experiment was implemented by running the Actor-Critic algorithm with 100 episodes for plotting its learning curve. Even though there are many fluctuation during episodes, the graph shows the convergence of Actor-Critic algorithm at about 70. Unlike other previous algorithms, Actor-Critic algorithm starts from positive values, meaning that it may affect the comparison it with other algorithms since the accumulated rewards can be affected by large negative values. For the Error bar in Figure 6 (right), the test was taken 100 trials (=10,000 episodes), and it shows the convergence of Actor-Critic algorithm. The average of the accumulated rewards in Actor-Critic was about 70.



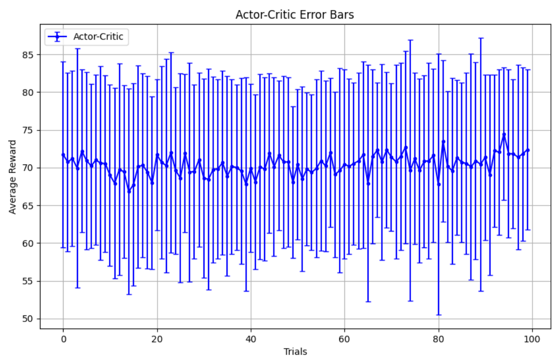
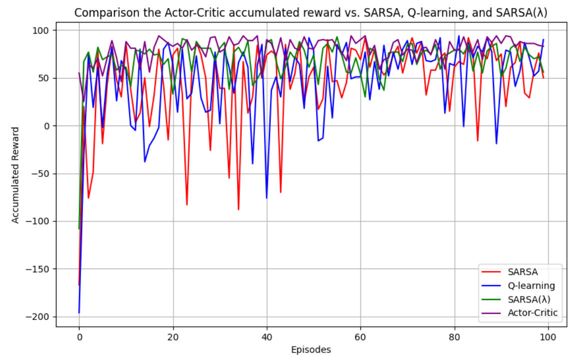


Figure 6: TD Actor Critic Learning Curve(left) and Error bar(right)

2. **(15 points)** Compare the accumulated reward at the end of 100 episodes with that of SARSA, Q-learning, and SARSA(λ). Discuss which algorithm performs better in this problem and why.

Accumulated reward Comparison all four algorithms above at the end of 100 episodes is in Figure 7. The left picture in Figure 7 is the result of merging all algorithms' learning curve, and the right one in Figure 7 shows accumulated reward comparison. In the figure (right), Actor-Critic algorithm has the most largest rewards, 8042.0, compared to other algorithms. It starts from higher positive value 30 than others, which have negative value less than 0, and takes the first place at the end of the 100 episodes. The lowest one is SARSA algorithm, which is 4550.0. Q-learning and SARSA(λ) have 4781.0 and 6892.0 respectively. From the result, it indicates that starting positive value rewards and keeping steady state may bring out the largest value and can take the first place.

One of the reasons why this happen is that TD Actor-Critic algorithm converges faster than others, updating both the policy and value function simultaneously. TD Actor-Critic uses state-value function with functional approximation to reduce variance. The policy learning seems more flexible than others because of fast calculation of the value function.



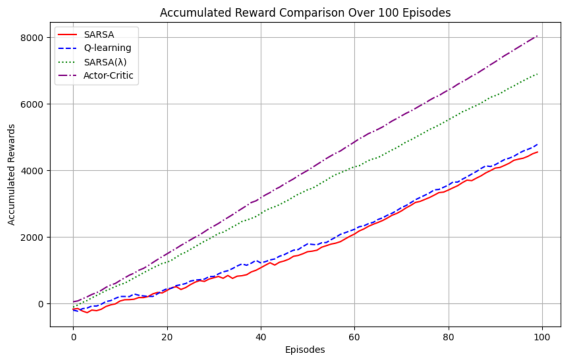


Figure 7: Accumulated Reward Comparison Actor Critic with SARSA, Q-Learning, and SARSA(λ)