HW 3

EECE 571N – Sequential Decision-Making (EECE 571N)

Instructor: Cyrus Neary
Due: 2025-11-10 at 23:59 PT

Name: Dominic Klukas **Student Number:** 64348378

Instructions

Submit a single PDF to Canvas. Please include your name and student number at the top of that PDF. For all questions, show your work and clearly justify your steps and thinking. Please feel free to include any code as an attachment at the end of the PDF. State any assumptions. Unless otherwise specified, you may collaborate conceptually but must write up your own solutions independently.

Grading

Points for each part are indicated. The total number of achievable points is 100. Partial credit is available for incorrect answers with clear reasoning.

Starter Code

This assignment asks you to implement certain ideas and algorithms in Python. Please use the starter code provided at:

`https://github.com/cyrusneary/UBC_EECE571N_fall_2025/blob/main/Homework-starter-code/HW3`

Fill in the missing `TODO` comments to complete the assignment. Insert your figures as images to the relevant questions of your submission PDF document, and snippets of the relevant portions of your code.

Questions

1. [5] **Distinguishing on-policy and off-policy RL algorithms.** In your own words, explain the difference between on-policy and off-policy learning. Give one example of an algorithm from each category, and describe briefly how the data they use to update their value estimates differs.

   Solution. On policy learning both involve improving a policy's decisions with respect to the actions it takes at a given state from data about the predicted gains in value that a policy will have if it changes it's choices at a given state, but on-policy learning makes these predictions based off of data collected by running the policy that is being improved in the real world, wheras off-policy learning makes the predictions based off of data collected from a different policy, evaluating it's decisions instead to determine if the policy that is learning should change it's actions.

   For the two examples, the traditional Monte-Carlo general policy improvement aglorithm is on policy, and the Monte-Carlo policy improvement algorithm that uses importance sampling

is an example of an off-policy algorithm. I will go through my understanding of how they work below:

Traditional monte-carlo method: for each state $s \in S$ and action $a \in A(s)$, runs rollouts directly on the policy, and uses them to approximate $Q(s,a)$. Then, the policy updates with $\pi_{\text{next}}(s) = \text{argmax}_a Q(s,a)$, and then runs rollouts again to improve this next policy.

The off-policy version of the Monte Carlo uses importance sampling. It modifies the estimation of the expectation of trajectories to match the probability that the learning policy, rather than the policy that the trajectory was actually sampled from. Essentially, the rollouts estimate the quantity $Q_\pi(s,a) = \sum_\tau G_t Pr(\tau|(s_0,a_0) = (s,a), \pi)$. We can compute the learning expectation, then, with

$$\begin{aligned} Q_{\pi_{\text{learn}}}(s,a) &= \sum_\tau G \cdot Pr(\tau|(s_0,a_0) = (s,a), \pi_{\text{learn}}) \\ &= \sum_\tau G \cdot \frac{Pr(\tau|(s_0,a_0) = (s,a), \pi_{\text{learn}})}{Pr(\tau|(s_0,a_0) = (s,a), \pi_{\text{sampled}})} Pr(\tau|(s_0,a_0) = (s,a), \pi_{\text{sampled}}). \end{aligned}$$

However, the expressions $Pr(\ldots)$ are products of the transition probabilities and the probabilities that the policies take an action at each state, so when we take the ratio of two for the same trajectory but with different policies, we just can can compute this ratio without knowledge of the dynamics of the function, and we call it $\rho_\tau = \prod_{t=1}^{T_{\tau_{\text{final}}}} \frac{\pi_{\text{learning}}(a_{\tau_t}|s_{\tau_t})}{\pi_{\text{sampled}}(a_{\tau_t}|s_{\tau_t})}$. With sloppy notation I have denoted a_{τ_t} and s_{τ_t} to be the action and state respectively taken in trajectory τ at time t. In practice, this true expectation is sampled by doing lots of rollouts and taking the average: $Q(s,a) \approx \frac{1}{N}\sum_{n=1}^N G_n$ for N trajectories. So then, to compute the importance sampled-estimate, we get: $Q(s,a) \approx \frac{1}{N}\sum_{n=1}^N \rho_{\tau_n} G_n$

2. [5] **Temporal difference learning.** In your own words, what makes Temporal-Difference (TD) learn- ing different from Monte Carlo methods? Why is TD learning considered important in reinforcement learning?

   Solution. TD learning is different from Monte Carlo methods because, instead of rolling out a trajectory or multiple trajectories in order to get an estimate for $V(s)$, or $Q(s,a)$ at a state $s \in S$, it rolls out the result of one action/state pair to get the reward r and future state s', and then combines this with the current value estimate $V(s')$, rather than continuing to roll out $\sum_{t=1}^{t_{final}} \gamma^t r_t$ for a whole trajectory.

   It is considered important in reinforcement learning because it combines the benefits of DP in having backpropagation without the requirement of knowing a transition function, which is the benefit of the Monte-Carlo method.

3. [10] **Creating a gridworld environment for RL training.** Implement an RL environment class representing a gridworld that follows the standard Gymnasium API for RL environments: `https://gymnasium.farama.org/index.html`. In Homework 1, you already implemented an MDP class modeling a gridworld environment. To help you get started with this question I have included a complete implementation of this GridworldMDP class in the starter code repository. To answer this question, please simply create a wrapper around this GridworldMDP class, by completing the starter code available here: `https://github.com/cyrusneary/UBC_EECE571N_fall_2025/blob/main/Homework-starter-code/HW3/gridworld_gym_env.py`. Include a snippet of the code implement- ing this Gymnasium environment wrapper in your submission file.

The initialization function of your GridworldEnv class should take the height and width of the grid as inputs, as well as the initial state, goal location, sink location(s), wall positions, slip probabil- ity, discount factor γ, and reward function parameters. Figure 1 illustrates an example gridworld environment that we will be using in several of the following questions.

Solution Initialization:

```
            # TODO: Define action and observation spaces and expose nS and nA
        self.state = self.mdp.init
        self.nS = self.mdp.nS
        self.nA = self.mdp.nA
```

Reset function. Added an option to set the starting state.

```
    def reset(self, seed=None, options: dict = None):
        """
        Reset the environment to the initial state.
        Arguments:
                options: "state", integer in [0, nS),
                the index of the desired starting state
        Returns:
          obs: the observation corresponding to the reset initial state
            (in this case just return the initial state index itself)
          info: additional info (empty dict in this case)
        """
        if options is not None:
            self.state = options["state"]
        else:
            self.state = self.mdp.init

        observation = self.state
        info = {}
        return observation, info
```

```
    def step(self, action):
        """
        Take an action in the environment.
        Arguments:
            action: the action to take (integer in [0, nA-1])
        Returns:
          next_state: the next state after taking the action
          reward: the reward received
          done: whether the episode has ended
          info: additional info (empty dict here)
        """
        # TODO: Implement step method (hint, make use of self.mdp.p(state, action))
        # Get the transitions for the state action pair
        transitions = self.mdp.P(self.state, action)

        # Choose the transition
        probabilities = np.array([t.prob for t in transitions])
        probabilities = probabilities/np.sum(probabilities)
        chosen_transition = np.random.choice(transitions, p=probabilities)
```

```
20
21          # Get the results
22          next_state = chosen_transition.next_state
23          self.state = next_state
24          reward = chosen_transition.reward
25          done = chosen_transition.done
26          info = {}
27          return next_state, reward, done, info
```

4. [20] **Monte Carlo estimation of $V^\pi(s)$.** Using the gridworld environment from the previous ques- tion, implement the "First-Visit Monte Carlo Prediction" algorithm that we saw in class. Use the starter code file available at : `https://github.com/cyrusneary/UBC_EECE571N_fall_2025/blob/main/Homework-starter-code/HW3/monte_carlo_code.py`.

   To test your implementation, use it to predict the value function corresponding to the optimal policy for the gridworld illustrated in Figure 1. Run the algorithm for num_episodes = 10000 episodes. The gridworld parameters (e.g., $\gamma = 0.9$, slip probability $p = 0.05$), as well as the optimal policy for the gridworld, are already available at the bottom of the provided starter code file.

   (a) Include a plot of the RMSE of the estimated value function $\hat{V}_n(s)$, as a function of the number episodes n sampled in the environment. Here, the RMSE should be defined as

   $$\text{RMSE}(n) = \sqrt{\sum_{s \in S} (\hat{V}_n(s) - V^*(s))^2},$$

   where $V^*(s)$ is the value of the optimal policy in state s.

   (b) Use the `mdp.plot_values()` function (provided in the GridworldMDP starter code) to visualize the error $\text{err}(s) = |\hat{V}_n(s) - V^*(s)|$ of the estimated value function after $n = 10000$ episodes separately for each state in the gridworld. Also use it to visualize the number of visits $N_n(s)$ to each state s after $n = 10000$ episodes.

   (c) In your own words, describe what you notice about the plots from part b). In which states is $\text{err}(s)$ the highest? What do you notice about the values of $N_n(s)$ in the corresponding states?

   (d) Include snippets of your Python code implementing the Monte Carlo prediction algorithm, and the generation/plotting of results.

   Solution.

   (a) Note: the problem called for 10,000 episodes. In order to compute the value at each state, I divided the 10,000 episodes by the number of states, and so the value of the first-visit monte carlo estimation at each state is the average of $10,000/n$ rollouts.

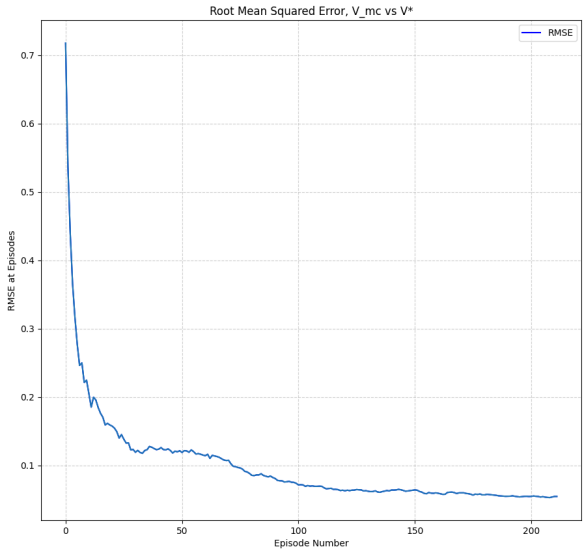
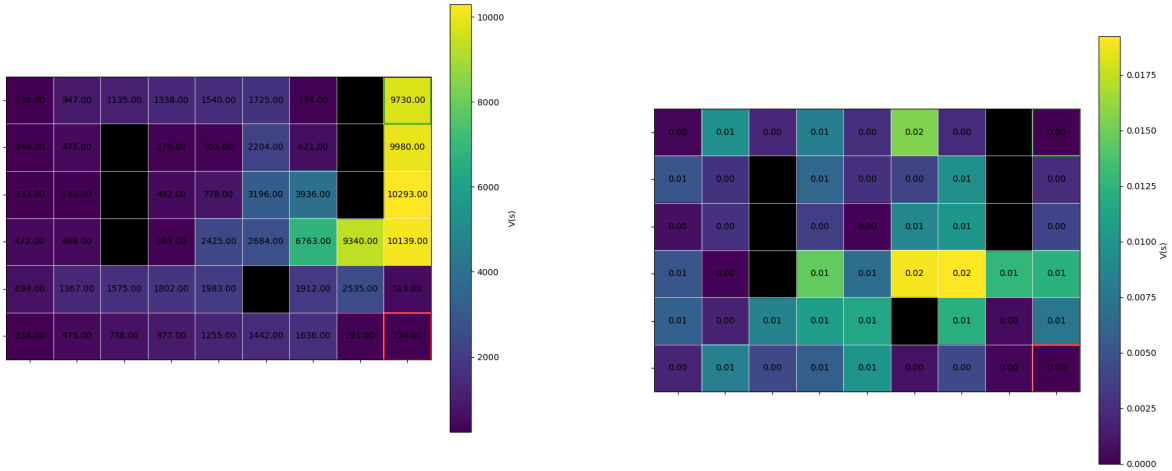


Figure 1: Root Mean Squared Error (RMSE) over episodes for the Monte-Carlo algorithm.



(a) Number of Visits at Each State for 10,000 Episodes

(b) L_1 Error of V_{MC} after 10,000 Episodes

Figure 2: State visit counts of Monte Carlo algorithm, and value estimation error after 10,000 episodes.

(b)

(c) After generating a few different plots, the error is highest in a seemingly random distribu-

tion of states. The states that get visited the most are the states that are in the path of the optimal policy for the greatest number of initial of starting states. However, since we are implementing the first-visit monte carlo prediction, every state gets the same number of samples: $\lfloor 10,000/19 \rfloor = 526$; even if some states get visited more from trajectories that start from other states or when the initial state gets revisited, this does *not* count as the start of another "run" to increase the samples considered for these oft visited states any more than the states that are visited often.

(d) Monte-Carlo algorithm code:

```
    num_eps_per_state = int(num_episodes / env.nS)
    N_list = []
    V_list = []
    N_s_list = np.zeros(env.nS)
    for s in range(env.nS):
        V_s_list = []
        V_s = 0
        for i in range(num_eps_per_state):
            env.reset(options={"state": s})
            states, actions, rewards = generate_episode(env, policy)
            for st in states:
                N_s_list[st] += 1 # This implies that we are interested in the
                        # total number of visits over all the Monte-Carlo runs
            discounts = gamma ** np.arange(len(rewards))
            G_i = np.sum(discounts * rewards)
            V_s += (G_i-V_s) / (i + 1)
            V_s_list += [V_s]
        V_list.append(V_s_list)
        N_list.append(N_s_list.copy())

    # Transform shape to (num_eps_per_state,nS)
    V_list = np.array(V_list).T
    N_list = np.array(N_list)
    return V_list, N_list
```

Code to graph the RMSE plot:

```
def plot_RMSE(V_list):
    RMSE = []
    for i in range(num_episodes_per_state):
        RMSE += [np.linalg.norm(V_list[i] - V_optimal)]
    plt.figure(figsize=(6, 4))
    plt.plot(range(num_episodes_per_state), RMSE, label='RMSE', color='blue')
    plt.xlabel('Episode Number')
    plt.ylabel('Root Mean Squared Error, V_mc vs V*')
    plt.title('RMSE at Episode')
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.legend()
    plt.tight_layout()
    plt.plot(RMSE)
    plt.show()
```

Code to call the Monte-Carlo algorithm and graph the plots:

```
    num_episodes = 10000
    V_list, N_list = mc_prediction_first_visit(env, policy_optimal, num_episodes, gamma)

    V_mc_final = V_list[-1]

    env.mdp.plot_values(np.abs(V_mc_final - V_optimal), annotate=True)
    env.mdp.plot_values(N_list[-1], annotate=True)
    plot_RMSE(V_list)
```

5. [20] **SARSA, On-Policy Control.** Using the same gridworld environment as the previous ques- tions, implement the "SARSA: On-Policy Control" algorithm that we saw in class. Use the starter code file available at: `https://github.com/cyrusneary/UBC_EECE571N_fall_2025/blob/main/Homework-starter-code/HW3/SARSA_code.py`.

   To test your code file, use it to learn an optimal policy and state-action value function in the gridworld environment from Figure 1. Once again, run the algorithm for num_episodes = 10000 episodes. Try each of the following learning rates: $\alpha = 0.01$, $\alpha = 0.05$, $\alpha = 0.15$.

   (a) Compare the plots of RMSE vs. number of episodes for each of the tested values of α. More specifically, let $\hat{Q}_n^\alpha(s,a)$ be the estimated value function after n episodes when the al- gorithm is using learning rate α. Define $\hat{V}_n^\alpha(s) = \arg\max_{a \in A} \hat{Q}_n^\alpha(s,a)$, and

   $$\text{RMSE}(n,\alpha) = \sqrt{\sum_{s \in S} (\hat{V}_n^\alpha(s) - V^*(s))^2}.$$

   Plot RMSE(n,α) as a function of n for $\alpha = 0.01$, $\alpha = 0.05$, and $\alpha = 0.15$.

   (b) In your own words, what do you observe? What happens to the algorithm as you change α? Why is this the case?

   (c) Include snippets of your Python code implementing the SARSA algorithm, and the genera- tion/plotting of results.

   Solution.

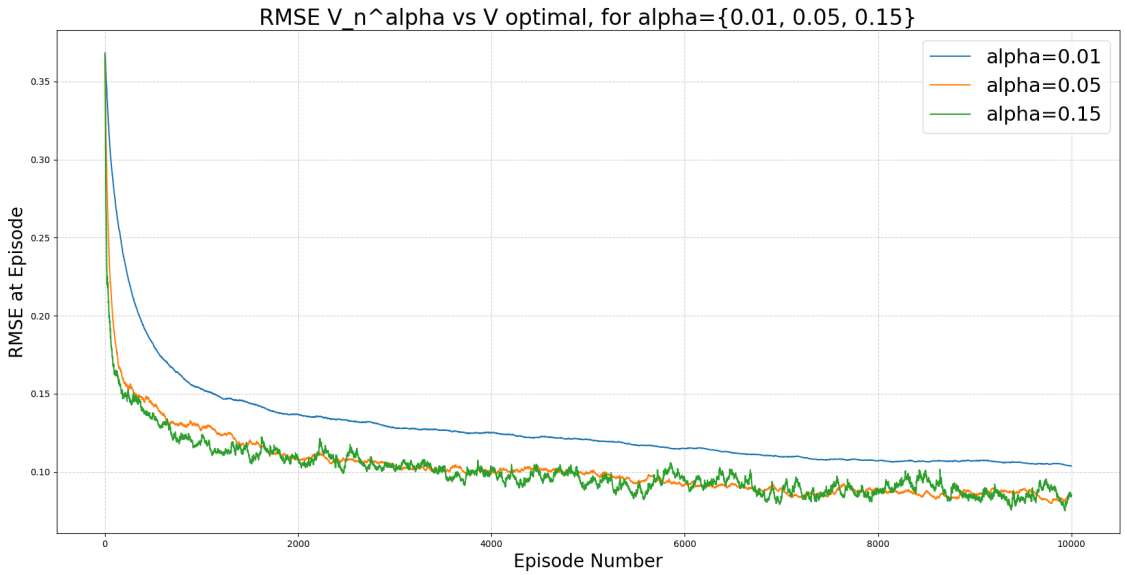


Figure 3: Root Mean Squared Error (RMSE) over episodes for the SARSA algorithm value function.

(a)

(b) The bigger the learning rate, the more quickly the RSME reduces, but the noisier the signal is. Close to the end, $\alpha = 0.05$ has lower average RMSE over the last 1000 episodes than $\alpha = 0.15$, because it has lower noise, so having too big of a learning rate can be deterimental to performance.

(c) Code implementing the SARSA algorithm:

```
    Q_list = [] # to store a copy of Q at the start of each episode
    N_visit_list = [] # to store a copy of N_visit at the start of each episode
    Q = np.zeros((env.nS, env.nA), dtype=np.float64)
    N_visit = np.zeros(env.nS, dtype=np.int32)

    def policy(Q_p, s, eps):
        nA = Q_p.shape[1]
        if np.random.random() < eps:
            return np.random.randint(nA)
        else:
            return np.argmax(Q_p[s])
    length = 0
    for _ in range(num_episodes):
        Q_list.append(Q.copy())
        N_visit_list.append(N_visit.copy())
        env.reset()
        done = False
        state = env.state
        states = [state]
        if _ %1000 == 0:
            print(f"We are at {_} with average episode length {length}", flush=True)
```

```
            length = 0
        length_ep = 0
        while not done:
            length_ep += 1
            N_visit[env.state] += 1
            action = policy(Q, env.state, epsilon)
            next_state, reward, done, info = env.step(action)
            next_action = policy(Q, next_state, epsilon)
            Q[state, action] += alpha*(reward + gamma*Q[next_state, next_action] - Q[state, ac
            states += [env.state]
            state= env.state
        length += (length_ep - length)/(_ - (_/1000)*1000 + 1)
    return Q_list, N_visit_list
```

Code to run the experiment/plot results:

```
    # SARSA experiment
    alpha_list = [0.01, 0.05, 0.15] # different learning rates to try
    num_episodes = 10000

    # TODO: Implement answers to homework questions.
    def RMSE_calculator(V_n, V_optimal):
        return np.sqrt(np.mean((V_n - V_optimal)**2))


    rmse_list = []
    n = np.arange(1, num_episodes+1)
    for alpha in alpha_list:
        Q_list, N_visit_list = sarsa(env,num_episodes,alpha,gamma,epsilon)
        V_list = [np.max(q, axis=1) for q in Q_list]
        rmse = np.array([RMSE_calculator(v, V_optimal) for v in V_list])
        plt.plot(n, rmse, label=f"alpha={alpha}")
    plt.xlabel('Episode Number', fontsize=20)
    plt.ylabel('RMSE at Episode', fontsize=20)
    plt.title('RMSE V_n^alpha vs V optimal, for alpha={0.01, 0.05, 0.15}', fontsize=24)
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.legend(fontsize=22)
    plt.tight_layout()
    plt.show()
```

6. [20] **Q-Learning.** Implement the tabular Q-learning algorithm we discussed in class. Use the starter code available at: `https://github.com/cyrusneary/UBC_EECE571N_fall_2025/blob/main/Homework-starter-code/HW3/Q_learning_code.py`. As a part of your implementation, please have the algorithm return both `Q_list` and `N_visit_list`. `Q_list` should be a list of numpy arrays, representing the estimated Q-function after each learning episode. Meanwhile, `N_visit_list` should also be a list of numpy arrays, each of these arrays should contain the visit counts for each state s in the gridworld, after each learning episode.

   To test your code file, use it to learn an optimal policy and state-action value function in the gridworld environment from Figure 1. Once again, run the algorithm for num_episodes = 10000 episodes. Try each of the following learning rates: $\alpha = 0.01$, $\alpha = 0.05$, $\alpha = 0.15$.

   (a) As described in part a) of the previous question, compare plots of the RMSE vs. number of episodes for each of the tested values of α.

(b) Use the results with learning rate $\alpha = 0.15$ to create two plots: the first should illustrate the error $\text{err}(s) = |\hat{V}_n^\alpha(s) - V^*(s)|$ of the estimated optimal value function after $n = 10000$ episodes; the second plot should illustrate the number of visits $N_n^\alpha(s)$ to each state s after $n = 10000$ episodes. Use the implementation of `mdp.plot_values()` provided in the GridworldMDP starter code to create these plots.

(c) Remove the wall at position (0,7) of the gridworld, re-run the Q-learning algorithm on this modified environment, and re-produce the plots of $\text{err}(s)$ and $N_n^\alpha(s)$ described in b).

(d) In your own words, what's the difference between the plots in parts b) and c)? In which states does the algorithm's value estimates have high error? Which states are visited more frequently by the algorithm? Why does removing the wall at (0, 7) change these properties, and what does this demonstrate about the algorithm's exploration process?

(e) Include snippets of your Python code implementing the Q-learning algorithm, and the genera- tion/plotting of results.

Solution.

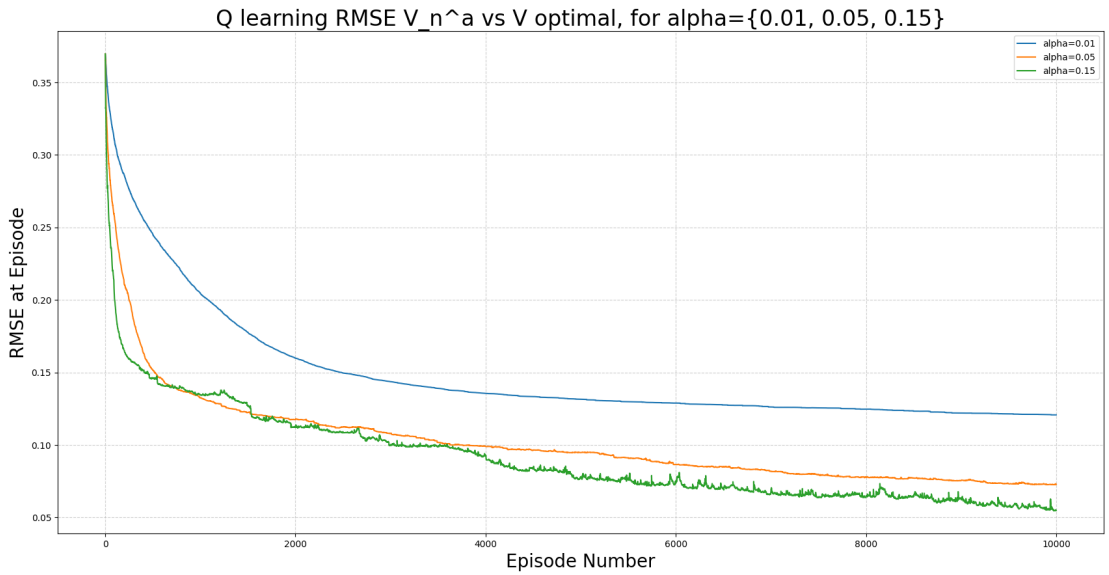
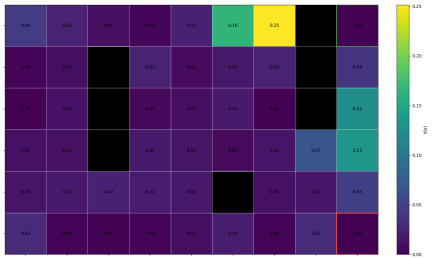


Figure 4: Root Mean Squared Error (RMSE) over episodes for the Q learning algorithm.

(a)

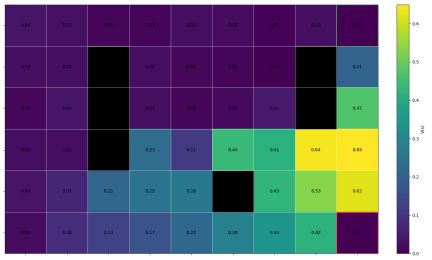


(a) L_1 Error of V_{MC} after 10,000 Episodes

(b) Number of Visits at Each State for 10,000 Episodes

Figure 5: State visit counts of the Q-learning algorithm, and value estimation error after 10,000 episodes. $\alpha = 0.15$.

(b)



(a) L_1 Error of V_{MC} after 10,000 Episodes

(b) Number of Visits at Each State for 10,000 Episodes

Figure 6: State visit counts of the Q-learning algorithm, and value estimation error after 10,000 episodes. Wall at $(0, 7)$ removed. $\alpha = 0.15$.

(c)

(d) The plots differ because removing the wall at (0, 7) in part c) changes the optimal path, allowing a shorter route to the goal. As a result, the algorithm spends most of its time near this new optimal path. In both cases, states close to the optimal path are visited most frequently, while states farther away—especially those near obstacles—show higher value-estimation error. Removing the wall changes the visitation pattern and reduces errors along the new path, demonstrating that the algorithm's exploration is heavily guided by the discovered optimal route rather than uniform state coverage. Indeed, in part b), the state (0,6) is rarely visited precisely because it is far from the optimal path, but once the optimal path passes through (0, 6), it becomes very frequently visited.

(e) Q-learning algorithm:

```
    Q_list = [] # to store a copy of Q at the start of each episode
    N_visit_list = [] # to store a copy of N_visit at the start of each episode

```

```
    Q = np.zeros((env.nS, env.nA), dtype=np.float64) # action-value function estimate
    N_visit = np.zeros(env.nS, dtype=np.int32) # state visit counts

    def policy(Q_p, s, eps):
        nA = Q_p.shape[1]
        if np.random.random() < eps:
            return np.random.randint(nA)
        else:
            return np.argmax(Q_p[s])
    length = 0
    for _ in range(num_episodes):
        Q_list.append(Q.copy())
        N_visit_list.append(N_visit.copy())
        env.reset()
        done = False
        state = env.state
        states = [state]
        if _ %1000 == 0:
            print(f"We are at {_} with average episode length {length}", flush=True)
            length = 0
        length_ep = 0
        while not done:
            N_visit[env.state] += 1
            action = policy(Q, env.state, epsilon)
            next_state, reward, done, info = env.step(action)
            Q[state, action] += alpha*(reward + gamma*np.max(Q[next_state, :]) - Q[state, acti
            states += [env.state]
            state= env.state

            #For Logging
            length_ep += 1
        length += (length_ep - length)/(_ - (_/1000)*1000 + 1)
    return Q_list, N_visit_list

```

The RMSE plotting code is virtually identical to SARSA RMSE plotting code. Plotting code, for grid maps:

```
    num_episodes = 10000
    alpha = 0.15
    Q_list, N_visit_list = q_learning(env,num_episodes,alpha,gamma,epsilon)
    V_list = [np.max(q, axis=1) for q in Q_list]
    print(np.array(V_list).shape,flush=True)
    print(np.array(N_visit_list).shape,flush=True)
    V_mc_final = V_list[-1]


    env.mdp.plot_values(np.abs(V_mc_final - V_optimal), annotate=True)
    env.mdp.plot_values(N_visit_list[-1], annotate=True)
```

7. [20] **Model-Based RL: Dyna-Q.** Implement the Dyna-Q algorithm that we discussed in class. Use the starter code available at: `https://github.com/cyrusneary/UBC_EECE571N_fall_2025/blob/main/Homework-starter-code/HW3/dyna_q_code.py`.

To test your code file, use it to learn an optimal policy and state-action value function in the gridworld environment from Figure 1. Run the algorithm for num_episodes = 10000 episodes. Use a single fixed learning rate of $\alpha = 0.15$. Try each of the following values for nplan, the number of "planning" steps to take per environment step: nplan = 0, nplan = 5, nplan = 50.

(a) Similarly to as in part a) of the previous two questions, compare plots of the RMSE vs. number of episodes for each of the tested values of nplan.

(b) Change the environment slip probability to $p = 0.3$, re-run the algorithm on this modified environment, and re-produce the RMSE plots described in part a).

(c) In your own words, how does the value of nplan affect the algorithm performance? What effect does increasing the slip probability in the environment have on this performance? Why is this the case?

(d) Include snippets of your Python code implementing the SARSA algorithm, and the genera- tion/plotting of results.

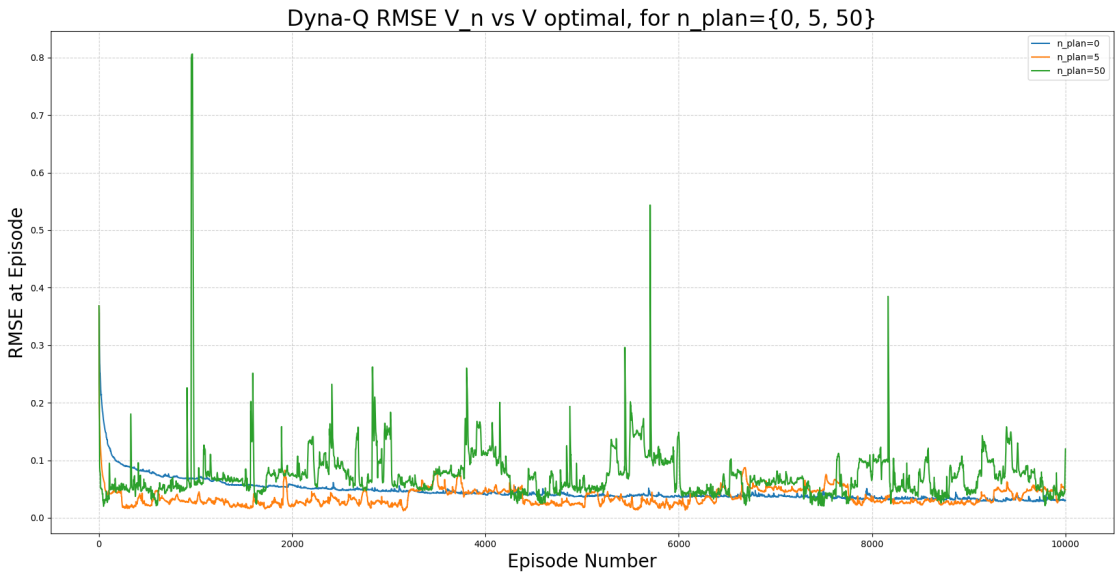


Figure 7: Root Mean Squared Error (RMSE) over episodes for the Dyna-Q algorithm given different planning models. $p = 0.05$.

(a)

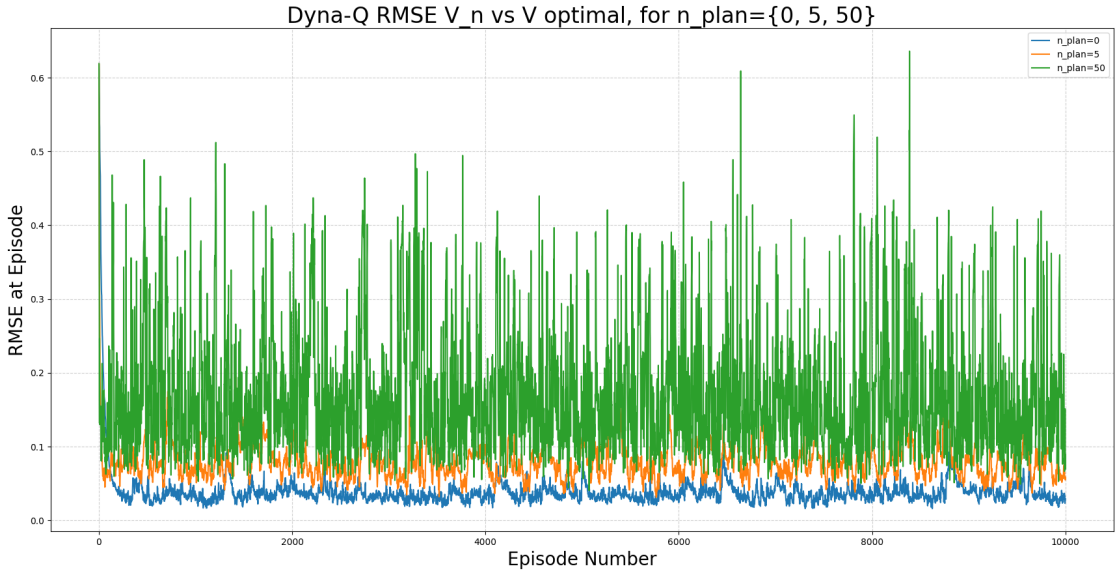


Figure 8: Root Mean Squared Error (RMSE) over episodes for the Dyna-Q algorithm given different planning models. $p = 0.3$.

(b)

(c) n_{plan} improves the algorithm performance drastically if it's model of the environment is accurate. We can see that when $p = 0.05$, in part a), we have that for both $n_{\text{plan}} = 5$ and $n_{\text{plan}} = 50$, the algorithm learns the optimal values instantly. However, eventually, the algorithm updates it's model with too many slip probabilities, and so seriously miscalculates its error values. The bigger nplan is, and the bigger p is, the more susceptible this algorithm is to such errors.

(d) Dyna-Q algorithm code:

```
    Q_list = []

    Q = np.zeros((env.nS, env.nA), dtype=np.float64)

    model = {}   # (s,a) -> list of (r, s_next, done)
    seen_keys = []

    def policy(Q_p, s, eps):
        nA = Q_p.shape[1]
        if np.random.random() < eps:
            return np.random.randint(nA)
        else:
            return np.argmax(Q_p[s])
    length = 0
    for _ in range(num_episodes):
        Q_list.append(Q.copy())
        env.reset()
```

```
        done = False
        state = env.state

        # logging code
        if _ %1000 == 0:
            print(f"We are at {_} with average episode length {length/1000}", flush=True)
            length = 0
        length_ep = 0

        while not done:
            action = policy(Q, env.state, epsilon)
            next_state, reward, done, info = env.step(action)
            Q[state, action] += alpha*(reward + gamma*np.max(Q[next_state, :]) - Q[state, acti

            if (state, action) not in model:
                seen_keys.append((state,action))
            model[(state, action)] = (reward, next_state, done)

            state = env.state
            for i in range(n_planning_steps):
                key = random.choice(seen_keys)
                state_p, action_p = key
                reward_p, next_state_p, irrelevant = model[key]
                Q[state_p, action_p] += alpha*(reward_p + gamma*np.max(Q[next_state_p, :]) - Q

            # Logging Code
            length_ep += 1
        length += length_ep

    return Q_list
```

Code for plotting:

```
    # Dyna-Q experiment
    n_list = [0, 5,50] # different n_planning_steps to try
    alpha = 0.15
    num_episodes = 10000


    # TODO: Implement answers to homework problems.
    def RMSE_calculator(V_n, V_optimal):
        return np.sqrt(np.mean((V_n - V_optimal)**2))


    rmse_list = []
    n = np.arange(1, num_episodes+1)
    for n_planning in n_list:
        Q_list = dyna_q(env, num_episodes, alpha, n_planning, gamma, epsilon)
        V_list = [np.max(q, axis=1) for q in Q_list]
        rmse = np.array([RMSE_calculator(v, V_optimal) for v in V_list])
        plt.plot(n, rmse, label=f"n_plan={n_planning}")
    plt.xlabel('Episode Number', fontsize=20)
    plt.ylabel('RMSE at Episode', fontsize=20)
    plt.title('Dyna-Q RMSE V_n vs V optimal, for n_plan={0, 5, 50}', fontsize=24)
    plt.grid(True, linestyle='--', alpha=0.6)
```

```
23      plt.legend(fontsize=22)
24      plt.tight_layout()
25      plt.legend()
26      plt.show()
```