Programming Assignment 3

SMDP and Intra Option Q-Learning

Gautham Govind A - EE19B022
Vishnu Vinod - CS19B048

CS6700
Reinforcement Learning

Department of Computer Science and Engineering
IIT Madras

Contents

1 CODE FOLDER

The entirety of the code written for this assignment as well as the output files containing all plots and notebooks are present in this Google Drive Link: link

2 Introduction

In this assignment, we implement SMDP and Intra-Option Q Learning on the Taxi-v3 environment. The agent will learn to make decisions at different levels of abstraction by learning a set of options, which are sequences of actions that can be executed without receiving a new observation or reward. Then, the agent will learn to use these options to improve its decision-making process. This allows the agent to make more efficient and effective decisions by taking into account the long-term goals and breaking them down into smaller sub-goals.

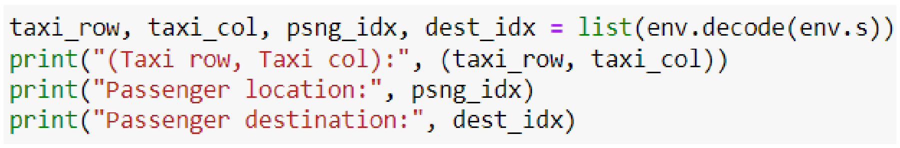
This implementation showcases how advanced reinforcement learning techniques can be used to solve complex problems efficiently and effectively. The resulting agent will be able to navigate the grid world, pick up and drop off passengers, and maximize its overall reward by using a combination of high-level and low-level decision-making processes.

3 Experimental Setup

3.1 Environment description

We make use of the ”Taxi-v3” environment provided by Gym. The problem is to transport a passenger from a source location to a destination location using a cab as fast as possible.This problem is modelled as a 5x5 gridworld, with 4 dedicated locations for pick-up and drop-off. This leads to a total of 500 possible states which are encoded as integers from 0 to 499. The environment also comes with 6 actions: move south, move north, move east, move west, pick-up, drop-off. These actions are encoded by integers from 0 to 5 in the respective order.

An important method which will be of great use to us is the decode method of the environment which helps you convert the integer-indexed state to more tangible quantities. We use the function as follows:



```
taxi_row, taxi_col, psng_idx, dest_idx = list(env.decode(env.s))
print("(Taxi row, Taxi col):", (taxi_row, taxi_col))
print("Passenger location:", psng_idx)
print("Passenger destination:", dest_idx)
```

Figure 1: Decoding the State of the Environment

`taxi_row` gives the row in which the taxi is currently located. `taxi_col` gives the column in which the taxi is currently located. `psng_idx` corresponds to the locations of the passenger and `dest_idx` corresponds to the destinations of the passenger as per the following table:

Value	`psng_idx`	`dest_idx`
0	**R**ed	**R**ed
1	**G**reen	**G**reen
2	**Y**ellow	**Y**ellow
3	**B**lue	**B**lue
4	**In** taxi	NA

We also note that the reward structure is as follows:

- **-1**: per step unless other reward is triggered
- **-10**: executing "pickup" and "drop-off" actions illegally
- **20**: for delivering a passenger

3.2 Option definition

We start by defining the options given in the problem statement. The problem statement asks us to define 4 options; one each for moving the taxi to one of the four designated locations (R, G, B, Y), executable when the taxi is not already there.

For defining an option we require three components: a policy, a termination condition and an initiation set. A description of these three for our case:

- **Policy**: The objective of the option is to move the taxi to a designated location. Since we are penalized with a -1 reward for every timestep, it makes sense to define the option such that it takes the least amount of time to reach the required location. We will keep this in consideration while designing policies for each option.
- **Termination condition**: In our case, this is rather straightforward as all the options terminate deterministically only on reaching the required location.
- **Initiation set**: The initiation set consists of all states apart from the target location. Therefore we use an if condition on the current state to check if the current state is a valid location to perform the option.

Next, we define the policy and termination conditions for the four options. We relegate the discussion of initiation sets to a later subsection. For the following discussion, taxi location (x, y) corresponds to the taxi being at row x and column y.

Move to R

We use the following conditions to define the policy (for the shortest path):

- Note that wherever possible, it is optimal to move left (west).
- Once you reach column 0, it is optimal to move up (north).
- If you are at (0, 2) or (1, 2), it is optimal to move down (south).
- If you are at (3, 1), (3, 3), (4, 1) or (4, 3), it is optimal to move up (north).

This option terminates on reaching (0, 0).

Move to Y

The policy is identical to the case of R except for the fact that you have to move down(south) in column 0 instead of up(north).

This option terminates on reaching (4, 0).

Move to G

We use the following conditions to define the policy:

- Once you reach column 4, it is optimal to move up (north).
- If you are at (0, 1) or (1, 1), it is optimal to move down (south).
- If you are at (3, 0), (3, 2), (4, 0) or (4, 2), it is optimal to move up (north).
- In all other cases, it is optimal to move right (east).

This option terminates on reaching (0, 4).

Move to B

We use the following conditions to define the policy:

- If you are at (4, 4), it is optimal to move left (west).
- Otherwise, once you reach column 3 or column 4, it is optimal to move down (south).
- If you are at (0, 1) or (1, 1), it is optimal to move down (south).
- If you are at (3, 0), (3, 2), (4, 0) or (4, 2), it is optimal to move up.
- In all other cases, it is optimal to move right (east).

This option terminates on reaching (4, 3).

We proceed to assign an integer value to each option as follows:

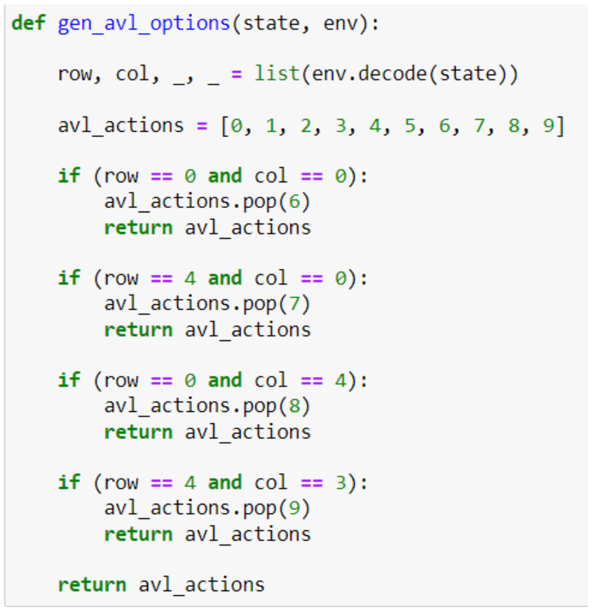
- 6 : Move to R
- 7 : Move to Y
- 8 : Move to G
- 9 : Move to B

Therefore, including both primitive actions and options, we have a total of 10 possible actions.

Initiation sets

We note that for each option, the initiation set is the set of all cells except for the target grid cell. For instance, for the option 'Move to R', initiation set consists of all cells except (0, 0).

We write a function `gen_avl_options()` which takes in a state as an argument and outputs all the possible options that can be executed from that state:

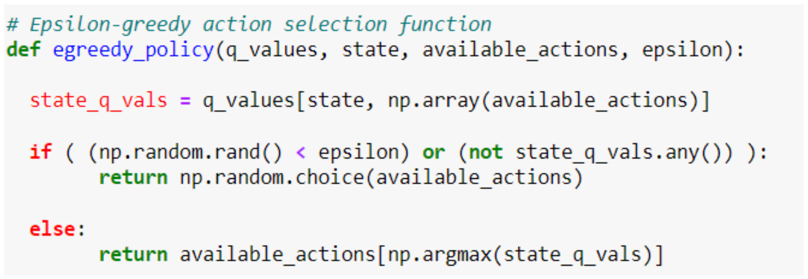


```python
def gen_avl_options(state, env):

    row, col, _, _ = list(env.decode(state))

    avl_actions = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    if (row == 0 and col == 0):
        avl_actions.pop(6)
        return avl_actions

    if (row == 4 and col == 0):
        avl_actions.pop(7)
        return avl_actions

    if (row == 0 and col == 4):
        avl_actions.pop(8)
        return avl_actions

    if (row == 4 and col == 3):
        avl_actions.pop(9)
        return avl_actions

    return avl_actions
```

Figure 2: Generating Available Options for each state

3.3 Exploration strategy

We make use of ϵ-greedy strategy for exploration. A slight difference from the usual approach here is that you should ensure that you only select from the list of available actions for each state. We ensure this by passing the list of available options which we obtain as the output of the gen_avl_options() functions as an input to ϵ-greedy and performing selection on this subset.



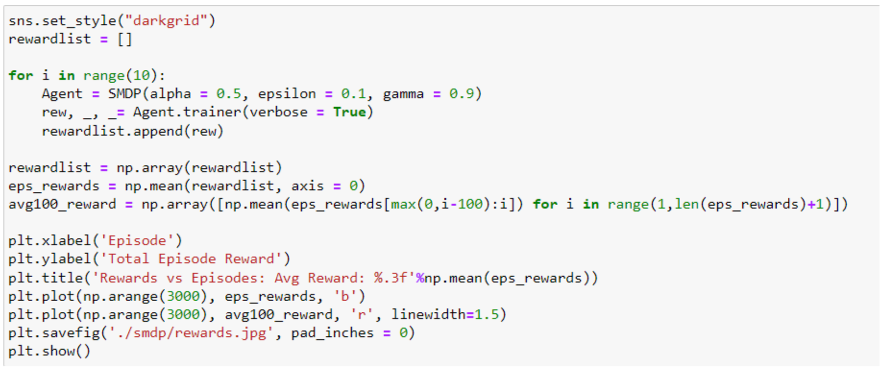
```python
# Epsilon-greedy action selection function
def egreedy_policy(q_values, state, available_actions, epsilon):

  state_q_vals = q_values[state, np.array(available_actions)]

  if ( (np.random.rand() < epsilon) or (not state_q_vals.any()) ):
        return np.random.choice(available_actions)

  else:
        return available_actions[np.argmax(state_q_vals)]
```

Figure 3: ϵ-Greedy Action Selection

3.4 Plotting Reward Curve

The reward curve is plotted across 10 runs and averaged with the number of episodes bound at 3000 for both SMDP and Intra-Option Q-Learning. The moving average across a window of 100 episodes is also plotted to give an idea of the average trends.

The code block for plotting reward curve is given below:

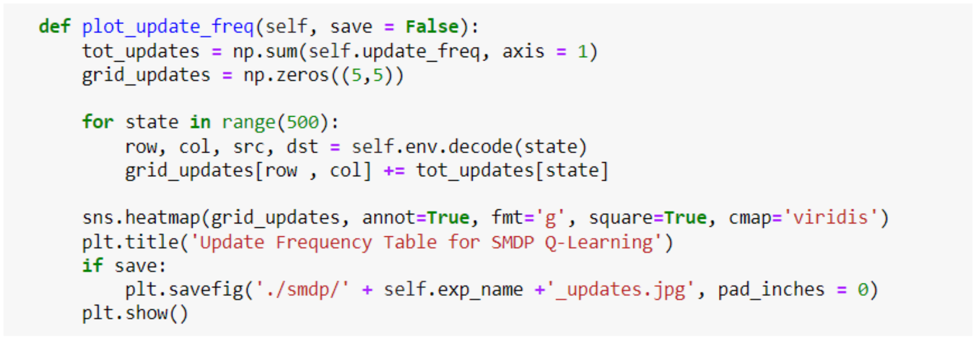


```
sns.set_style("darkgrid")
rewardlist = []

for i in range(10):
    Agent = SMDP(alpha = 0.5, epsilon = 0.1, gamma = 0.9)
    rew, _, _= Agent.trainer(verbose = True)
    rewardlist.append(rew)

rewardlist = np.array(rewardlist)
eps_rewards = np.mean(rewardlist, axis = 0)
avg100_reward = np.array([np.mean(eps_rewards[max(0,i-100):i]) for i in range(1,len(eps_rewards)+1)])

plt.xlabel('Episode')
plt.ylabel('Total Episode Reward')
plt.title('Rewards vs Episodes: Avg Reward: %.3f'%np.mean(eps_rewards))
plt.plot(np.arange(3000), eps_rewards, 'b')
plt.plot(np.arange(3000), avg100_reward, 'r', linewidth=1.5)
plt.savefig('./smdp/rewards.jpg', pad_inches = 0)
plt.show()
```

Figure 4: Plotting the Reward Curve

3.5 Plotting Update Frequency Tables

The update frequencies are stored in a 500×10 array with one value for every action and state combination. However in order to visualize them better we aggregate the update across all actions and simply plot the update frequency in the form of a table.



```
def plot_update_freq(self, save = False):
    tot_updates = np.sum(self.update_freq, axis = 1)
    grid_updates = np.zeros((5,5))

    for state in range(500):
        row, col, src, dst = self.env.decode(state)
        grid_updates[row , col] += tot_updates[state]

    sns.heatmap(grid_updates, annot=True, fmt='g', square=True, cmap='viridis')
    plt.title('Update Frequency Table for SMDP Q-Learning')
    if save:
        plt.savefig('./smdp/' + self.exp_name +'_updates.jpg', pad_inches = 0)
    plt.show()
```

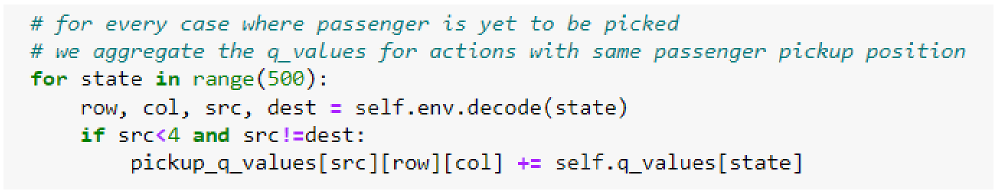
Figure 5: Plotting the Update Frequency Table

3.6 Plotting Q-Value Tables

The Q-values are stored in a 500×10 array with one value for every action and state combination. However in order to visualize them better we split the state set into 2:

- **Pick Up**: When `psng_idx` < 4 the passenger is not inside the taxi yet, the taxi must journey to the passenger location (`psng_idx`) and pick up the passenger
- **Drop Off**: When `psng_idx` $= 4$ the passenger is inside the taxi. The taxi must journey to the destination location `dest_idx` and drop off the passenger

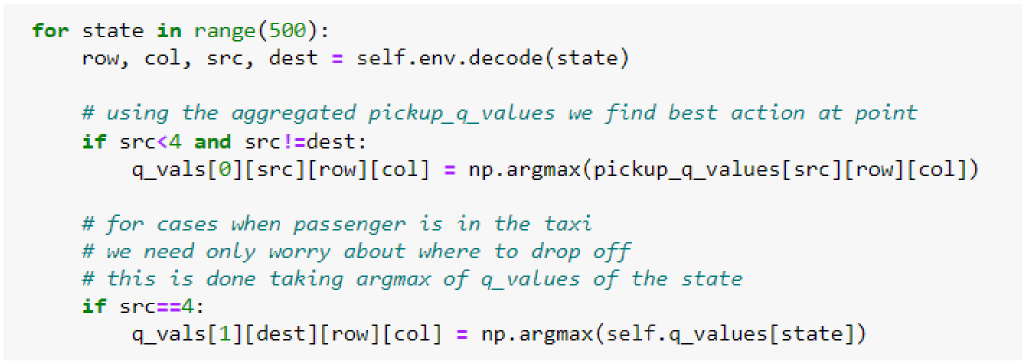
In order to form a cohesive understanding of how the taxi moves in the "pickup" phase, we aggregate the Q-values for each action and final destination belonging to a particular taxi location and pickup location of passenger. This is done using the following codeblock:



```python
# for every case where passenger is yet to be picked
# we aggregate the q_values for actions with same passenger pickup position
for state in range(500):
    row, col, src, dest = self.env.decode(state)
    if src<4 and src!=dest:
        pickup_q_values[src][row][col] += self.q_values[state]
```

Figure 6: Calculating Aggregate Q-Values for Pickup

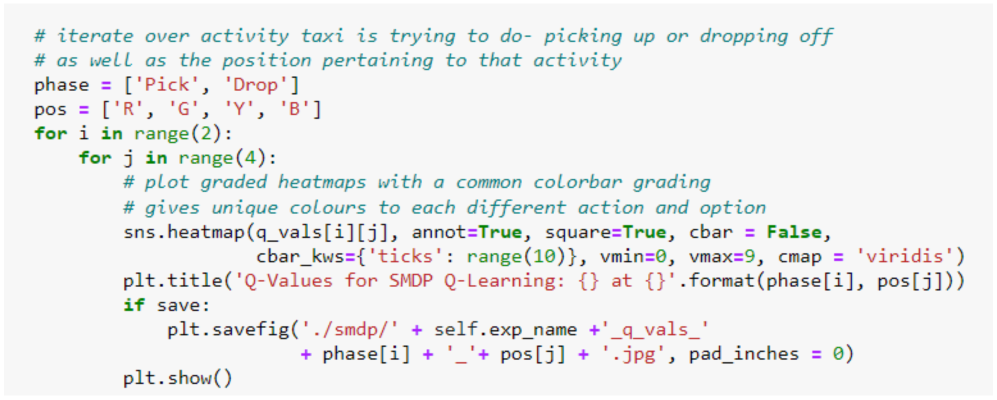
In order to find the favoured action of the taxi we calculate the action with the maximum Q-value amongst the available actions for each phase of the taxi movement as well as for various pickup and drop off locations.



```python
for state in range(500):
    row, col, src, dest = self.env.decode(state)

    # using the aggregated pickup_q_values we find best action at point
    if src<4 and src!=dest:
        q_vals[0][src][row][col] = np.argmax(pickup_q_values[src][row][col])

    # for cases when passenger is in the taxi
    # we need only worry about where to drop off
    # this is done taking argmax of q_values of the state
    if src==4:
        q_vals[1][dest][row][col] = np.argmax(self.q_values[state])
```

Figure 7: Calculating Action with highest Q-value

Once we have the required "best" actions we can now plot the actions chosen in the form of heatmaps for each of the 8 cases, with the phase being either pickup or drop off and the location of pickup/dropoff coming from the list $[R, G, Y, B]$.



```python
# iterate over activity taxi is trying to do- picking up or dropping off
# as well as the position pertaining to that activity
phase = ['Pick', 'Drop']
pos = ['R', 'G', 'Y', 'B']
for i in range(2):
    for j in range(4):
        # plot graded heatmaps with a common colorbar grading
        # gives unique colours to each different action and option
        sns.heatmap(q_vals[i][j], annot=True, square=True, cbar = False,
                    cbar_kws={'ticks': range(10)}, vmin=0, vmax=9, cmap = 'viridis')
        plt.title('Q-Values for SMDP Q-Learning: {} at {}'.format(phase[i], pos[j]))
        if save:
            plt.savefig('./smdp/' + self.exp_name +'_q_vals_'
                        + phase[i] + '_'+ pos[j] + '.jpg', pad_inches = 0)
        plt.show()
```

Figure 8: Generating Best Action Heatmaps

4 SMDP Q-Learning

Semi-Markov Decision Processes (SMDPs) extend Markov Decision Processes (MDPs) by introducing actions that can take multiple time-steps to complete, also known as options. An option is defined as a sequence of actions that terminates in a specific state. The state at which an option terminates is referred to as the ”goal” state.

SMDP Q-Learning is a value learning method which incorporates options by allowing for semi-markov processes unlike the vanilla Q-Learning approach which enforces an MDP assumption. It extends the classic Q-Learning algorithm to account for options. The key idea is to generalize primitive actions to more elaborate options so as to enable the agent to learn faster. At each time-step, the agent selects an action according to its Q-values for the current state. When an option is selected, the agent executes the option until it reaches the goal state. Upon reaching the goal state, the Q-value for the option is updated according to the reward obtained and the Q-values of the options in the goal state.

4.1 Update Equations

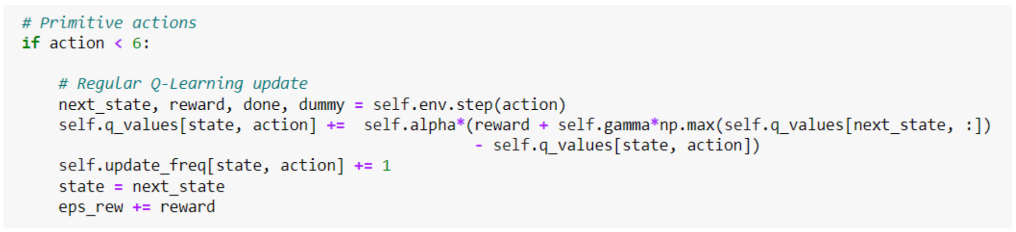
When a primitive action a is selected, the Q-value is updated according to the classic Q-Learning update equation:

$$\boxed{Q'(s,a) = Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{o'} Q(s',o') - Q(s,a)]}$$

where,

- $Q(o,s,a)$: Q-value of state s when action a is taken
- α: Learning Rate
- R: reward for choosing action a at state s
- γ: discount factor
- o': Next action/ option
- s': Next State

The code blocks for updates in case of primitive actions is shown below:



```
# Primitive actions
if action < 6:

    # Regular Q-Learning update
    next_state, reward, done, dummy = self.env.step(action)
    self.q_values[state, action] +=  self.alpha*(reward + self.gamma*np.max(self.q_values[next_state, :])
                                        - self.q_values[state, action])
    self.update_freq[state, action] += 1
    state = next_state
    eps_rew += reward
```

Figure 9: SMDP update for primitive actions

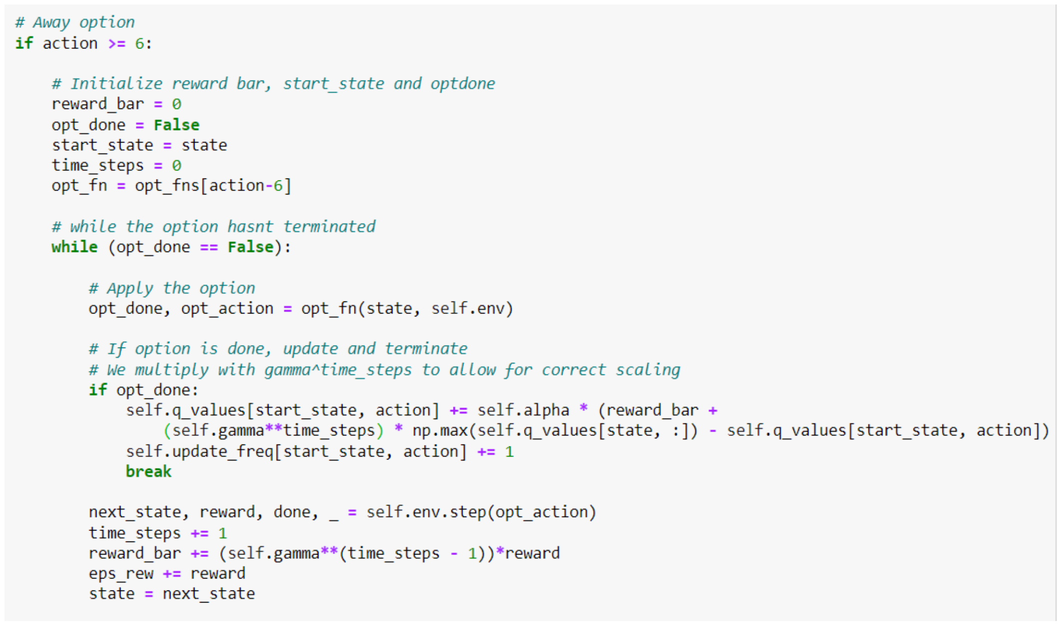
However the update equation for the Q-value of an option is as follows:

$$\boxed{Q'(o,s,a) = Q(o,s,a) + \alpha \cdot [R + \gamma^{\tau} \cdot \max_{o'} Q(o,s',o') - Q(o,s,a)]}$$

where,

- $Q(o, s, a)$: Q-value of option o in state s when action a is taken
- α: Learning Rate
- R: cumulative discounted reward across all actions taken as part of the option
- γ: discount factor while calculating return
- τ: Number of timesteps as part of option
- o': Next action/ option

The code blocks for updates in case of options selection is shown below:



```python
# Away option
if action >= 6:

    # Initialize reward bar, start_state and optdone
    reward_bar = 0
    opt_done = False
    start_state = state
    time_steps = 0
    opt_fn = opt_fns[action-6]

    # while the option hasnt terminated
    while (opt_done == False):

        # Apply the option
        opt_done, opt_action = opt_fn(state, self.env)

        # If option is done, update and terminate
        # We multiply with gamma^time_steps to allow for correct scaling
        if opt_done:
            self.q_values[start_state, action] += self.alpha * (reward_bar +
                (self.gamma**time_steps) * np.max(self.q_values[state, :]) - self.q_values[start_state, action])
            self.update_freq[start_state, action] += 1
            break

        next_state, reward, done, _ = self.env.step(opt_action)
        time_steps += 1
        reward_bar += (self.gamma**(time_steps - 1))*reward
        eps_rew += reward
        state = next_state
```

Figure 10: SMDP update for option selection

4.2 Hyperparameter Tuning

We carry out hyperparameter tuning on the hyperparameters like α (learning rate), and ϵ the control parameter for the ϵ-greedy action selection rule.

This metric used to evaluate the performance of a combination of hyperparameters is the **average reward**.

We test the following combinations of hyperparameters (total 20 configurations):

- α: 0.5, 0.1, 0.05, 0.01
- ϵ: 0.1, 0.05, 0.01, 0.005, 0.001

Note: It was observed during initial experimentation that when $\gamma = 0.99$ is selected the covergence of SMDP as well as the average rewards are comparable if not better than IntraOption, however for the sake of the assignment the given value of $\gamma = 0.9$ was used.

4.3 Results

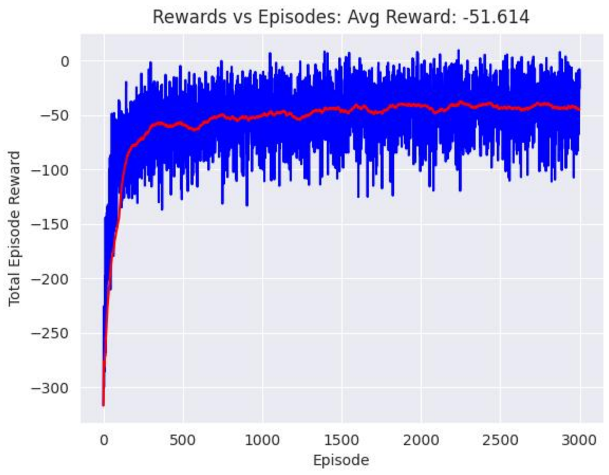


Figure 11: SMDP: Average reward plot

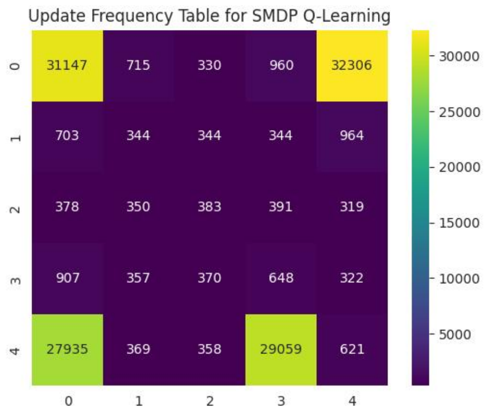


Figure 12: SMDP: Update frequency

From the average reward plot (averaged over 10 runs), we observe that the **rewards per episode increases as we expect.** This indicates that the agent is learning to solve the environment more efficiently. The red line indicates the running average over past 100 episodes, which increases as it should.

From the update frequency plot, we observe that the **agent largely makes use of options**. Since the defined options are **very precise and goal-oriented**, taking them would have lead to higher q-values and consequently higher update frequencies. Since primitive actions do not get updated when options are chosen, their update frequency remains low.

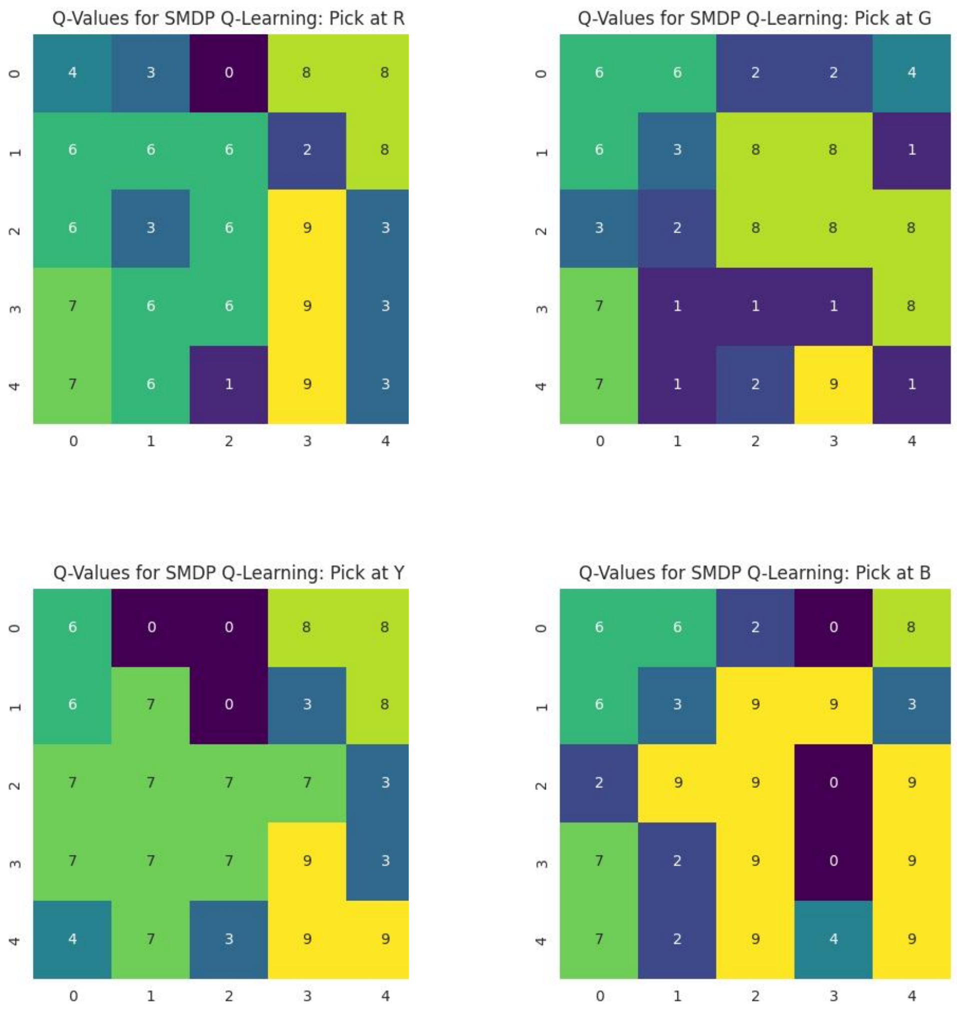


Figure 13: Actions/ Options with largest q-value for each grid cell

The four heatmaps displayed above indicate the preferred action for each grid cell based on the Q-value for the cases where the taxi is yet to pickup the passenger. These heatmaps provide insight into the policy learnt by the agent.

It can be seen that the taxi **picks up the passenger at the specified location in all cases** from the fact that action 4 (pick-up) is performed at the passenger locations. In other grid cells, the policy learnt is such that the **taxi moves closer to the passenger location** as can be seen from the fact that in many cells, the option to move to the passenger location is chosen (6 for R, 8 for G, 7 for Y and 9 for B). Even when other primitive actions/ options are chosen, they are generally such that they lead to the passenger location.

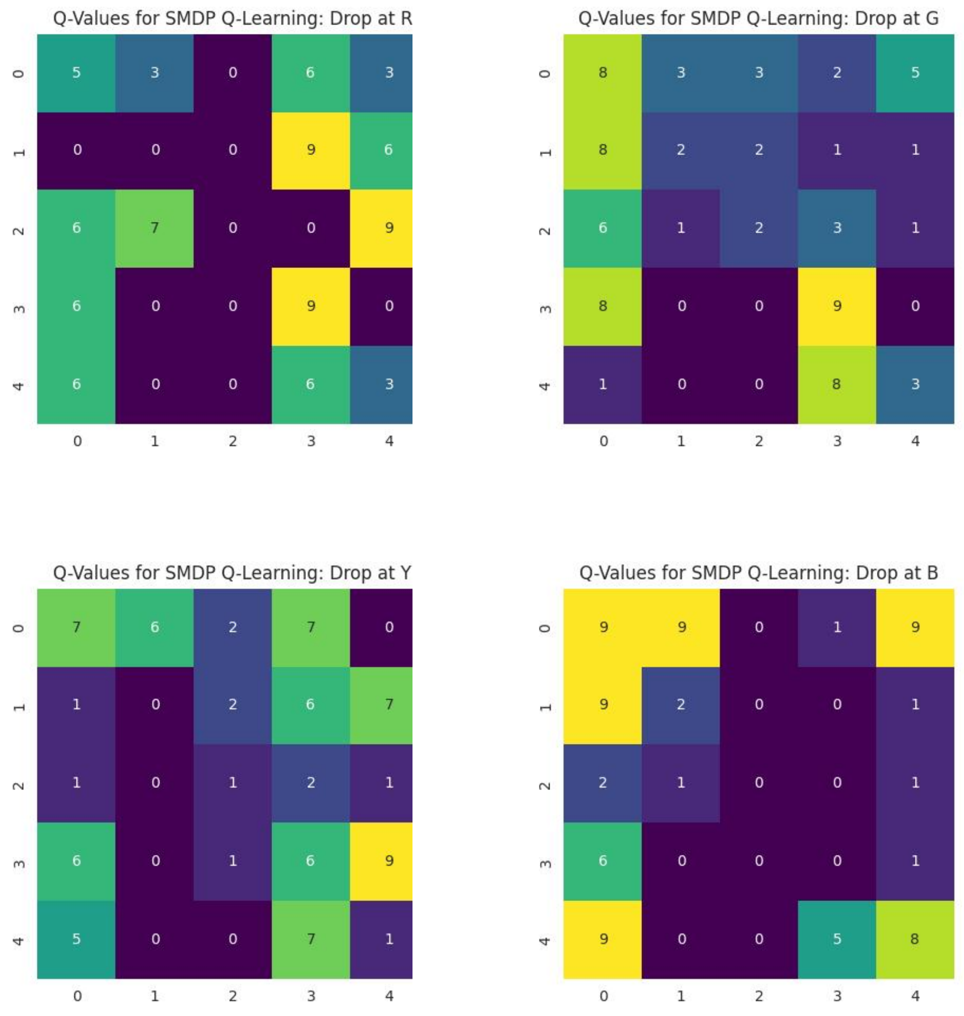


Figure 14: Actions/ Options with largest q-value for each grid cell

The four heatmaps displayed above indicate the preferred action for each grid cell based on the q-value for the cases where the taxi has picked up the passenger and is ready to drop the passenger off at the destination. These heatmaps provide insight into the policy learnt by the agent.

It can be seen that the taxi **drops off the passenger at the specified location in all cases** from the fact that action 5 (drop-off) is performed at the passenger locations. However, we note that the **performance in other grid cells are poorer when compared to the pick-up plots**. Though the required option is chosen in some cells, wrong actions/ options are chosen in several cases. This is probably because of the **slow convergence of the SMDP q-learning algorithm.**

5 Intra-Option Q-Learning

Intra-option q-learning is an extension of SMDP q-learning developed so as to obtain a more efficient alternative to SMDP q-learning. The key idea is the fact that SMDP methods require an option to be executed till termination which makes them relatively slow.

Intra-option methods essentially involve updating the q-values for all options that are compatible rather than for only the chosen option. Options which result in the same primitive action are termed compatible. Note that our definition of options also include primitive actions. As a result, a primitive action will be updated even if an option is chosen unlike the case of SMDP q-learning. Due to the much higher frequency of updates, Intra-option q-learning is expected to converge faster than conventional SMDP q-learning.

5.1 Update Equations

When a primitive action a is selected, like the SMDP case, we make the following update:

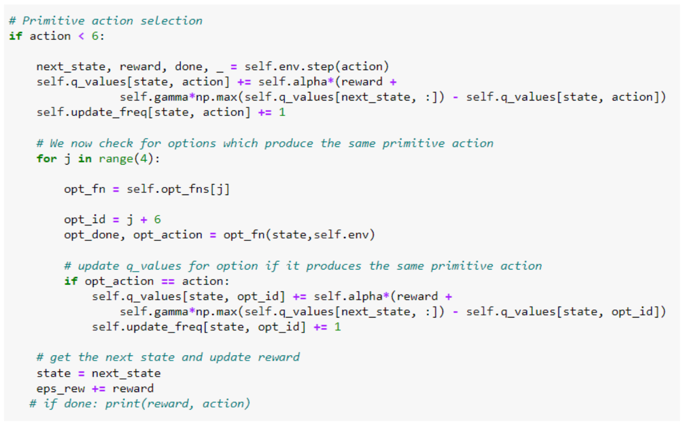
$$\boxed{Q'(s,a) = Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{o'} Q(s',o') - Q(s,a)]}$$

However, we now also perform the update:

$$\boxed{Q'(s,o) = Q(s,o) + \alpha \cdot [r + \gamma \cdot \max_{o'} Q(s',o') - Q(s,o)]}$$

$\forall\ o \in \tilde{\mathcal{O}}$ where $\tilde{\mathcal{O}}$ is the set of all options compatible with action a.

The code blocks for updates in case of actions selection is shown below:



```
# Primitive action selection
if action < 6:

    next_state, reward, done, _ = self.env.step(action)
    self.q_values[state, action] += self.alpha*(reward +
                self.gamma*np.max(self.q_values[next_state, :]) - self.q_values[state, action])
    self.update_freq[state, action] += 1

    # We now check for options which produce the same primitive action
    for j in range(4):

        opt_fn = self.opt_fns[j]

        opt_id = j + 6
        opt_done, opt_action = opt_fn(state,self.env)

        # update q_values for option if it produces the same primitive action
        if opt_action == action:
            self.q_values[state, opt_id] += self.alpha*(reward +
                self.gamma*np.max(self.q_values[next_state, :]) - self.q_values[state, opt_id])
            self.update_freq[state, opt_id] += 1

    # get the next state and update reward
    state = next_state
    eps_rew += reward
   # if done: print(reward, action)
```

Figure 15: IntraOp update for primitive actions

When an option o is selected, we perform the following updates **for all compatible options and primitive actions**:

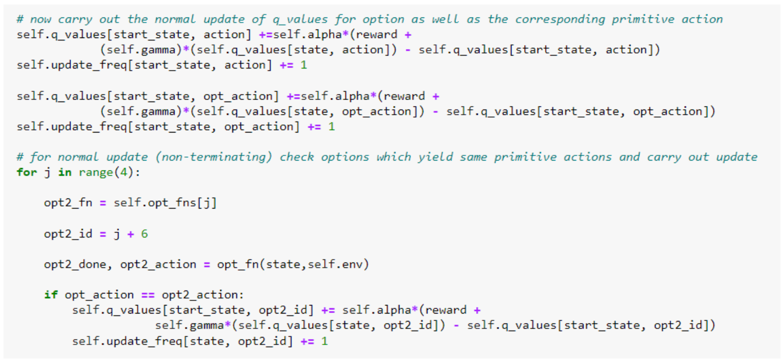
$$\boxed{Q'(s,o) = Q(s,o) + \alpha \cdot [r + \gamma \cdot Q(s',o) - Q(s,o)]}$$

if the option does not terminate at s' and

$$\boxed{Q'(s,o) = Q(s,o) + \alpha \cdot [r + \gamma \cdot \max_{o'} Q(s',o') - Q(s,o)]}$$

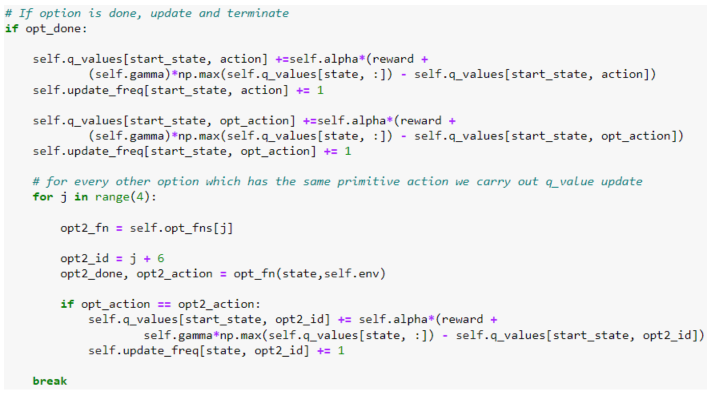
if the option terminates at s'.

The code blocks for updates in case of option selection is shown below:



```
# now carry out the normal update of q_values for option as well as the corresponding primitive action
self.q_values[start_state, action] +=self.alpha*(reward +
            (self.gamma)*(self.q_values[state, action]) - self.q_values[start_state, action])
self.update_freq[start_state, action] += 1

self.q_values[start_state, opt_action] +=self.alpha*(reward +
            (self.gamma)*(self.q_values[state, opt_action]) - self.q_values[start_state, opt_action])
self.update_freq[start_state, opt_action] += 1

# for normal update (non-terminating) check options which yield same primitive actions and carry out update
for j in range(4):

    opt2_fn = self.opt_fns[j]

    opt2_id = j + 6

    opt2_done, opt2_action = opt_fn(state,self.env)

    if opt_action == opt2_action:
        self.q_values[start_state, opt2_id] += self.alpha*(reward +
                    self.gamma*(self.q_values[state, opt2_id]) - self.q_values[start_state, opt2_id])
        self.update_freq[state, opt2_id] += 1
```

Figure 16: IntraOp update for options (Non-Terminating)



```
# If option is done, update and terminate
if opt_done:

    self.q_values[start_state, action] +=self.alpha*(reward +
            (self.gamma)*np.max(self.q_values[state, :]) - self.q_values[start_state, action])
    self.update_freq[start_state, action] += 1

    self.q_values[start_state, opt_action] +=self.alpha*(reward +
            (self.gamma)*np.max(self.q_values[state, :]) - self.q_values[start_state, opt_action])
    self.update_freq[start_state, opt_action] += 1

    # for every other option which has the same primitive action we carry out q_value update
    for j in range(4):

        opt2_fn = self.opt_fns[j]

        opt2_id = j + 6
        opt2_done, opt2_action = opt_fn(state,self.env)

        if opt_action == opt2_action:
            self.q_values[start_state, opt2_id] += self.alpha*(reward +
                    self.gamma*np.max(self.q_values[state, :]) - self.q_values[start_state, opt2_id])
            self.update_freq[state, opt2_id] += 1

    break
```

Figure 17: IntraOp update for options (Terminating)

5.2 Hyperparameter Tuning

We carry out hyperparameter tuning on the hyperparameters like α (learning rate), and ϵ the control parameter for the ϵ-greedy action selection rule.

This metric used to evaluate the performance of a combination of hyperparameters is the **average reward**.

We test the following combinations of hyperparameters (total 20 configurations):

- α: 0.5, 0.1, 0.05, 0.01
- ϵ: 0.1, 0.05, 0.01, 0.005, 0.001

The log files generated while carrying out this training are saved and present in the code folder.

5.3 Results

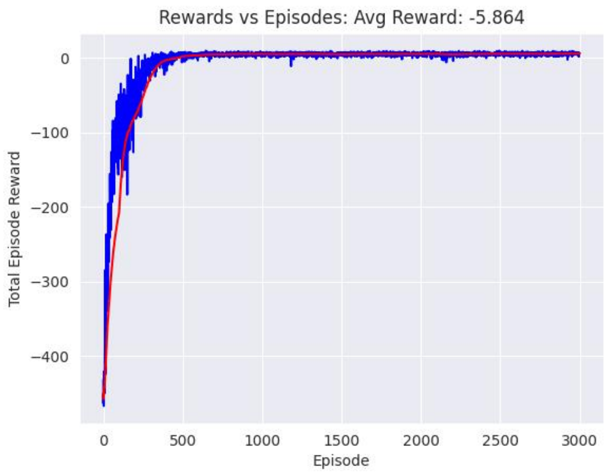
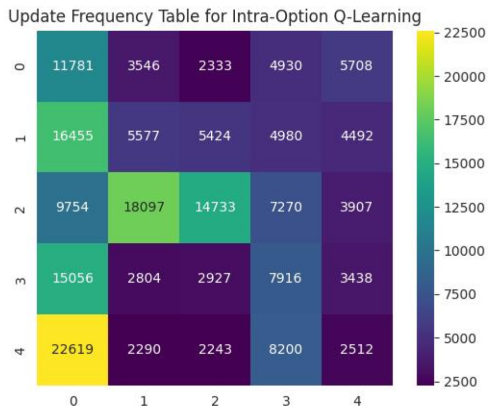


Figure 18: Intra-Option: Average reward plot



	0	1	2	3	4
0	11781	3546	2333	4930	5708
1	16455	5577	5424	4980	4492
2	9754	18097	14733	7270	3907
3	15056	2804	2927	7916	3438
4	22619	2290	2243	8200	2512

Figure 19: Intra-Option: Update frequency

From the average reward plot (averaged over 10 runs), we observe that the **rewards per episode increases as we expect.** The red line indicates the running average over past 100 episodes, which increases as it should. We also observe that the **avg reward is much higher than the SMDP case**.

From the update frequency plot, we observe that the **agent updates all grid cells rather uniformly**. This is in stark contrast with the case of SMDP q-learning where options were dominant. This could be attribute to the fact that in intra-option q-learning, all compatible actions/options are updated, therefore leading to a more uniform distribution.

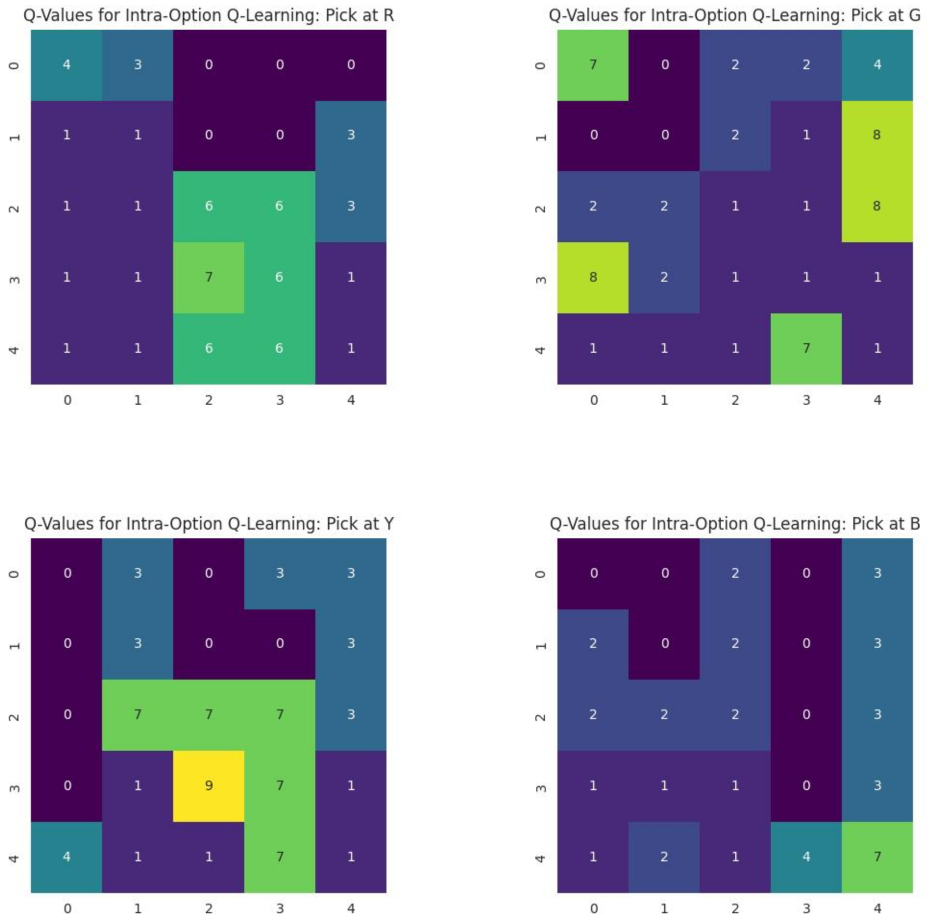


Figure 20: Actions/ Options with largest q-value for each grid cell

The four heatmaps displayed above indicate the preferred action for each grid cell based on the Q-value for the cases where the taxi is yet to pickup the passenger. These heatmaps provide insight into the policy learnt by the agent.

It can be seen that the taxi **picks up the passenger at the specified location in all cases** from the fact that action 4 (pick-up) is performed at the passenger locations. In other grid cells, the policy learnt is such that the **taxi moves closer to the passenger location**. However, the use of options directly is not as prevalent as in the SMDP case. This could be attributed to the fact that **all compatible actions are updated here.** Though options may not be used directly, compatible actions also lead to optimal results.

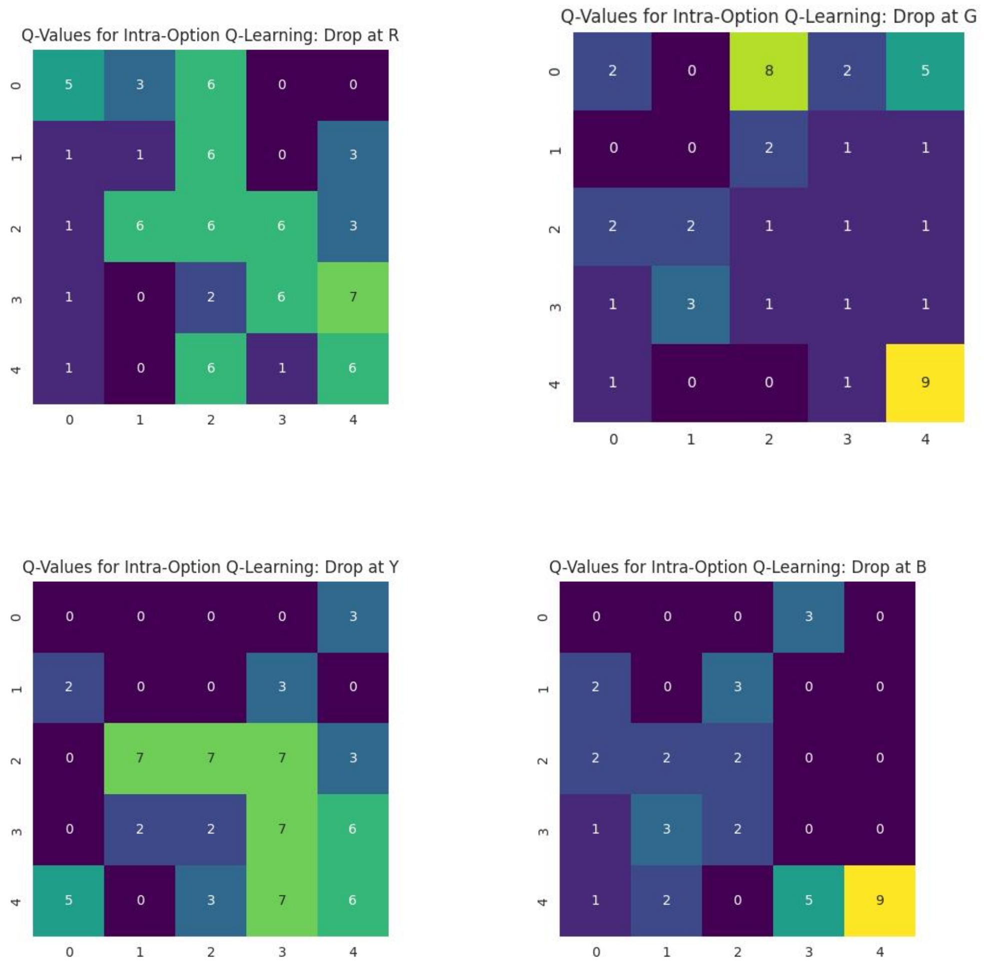


Figure 21: Actions/ Options with largest q-value for each grid cell

The four heatmaps displayed above indicate the preferred action for each grid cell based on the q-value for the cases where the taxi has picked up the passenger and is ready to drop the passenger off at the destination. These heatmaps provide insight into the policy learnt by the agent.

It can be seen that the taxi **drops off the passenger at the specified location in all cases** from the fact that action 5 (drop-off) is performed at the passenger locations. We also note that the **performance in other grid cells are as good as the performance in the pick-up plots**. This is in contrast to the SMDP case where the performance was much poorer compared to the pick-up case. This alludes to the fact that intra-option exhibits better performance thanks to the higher update frequency.

6 Comparing SMDP and Intra-Option Q-Learning

We first compare the rewards obtained in case of both SMDP and Intra-Option Q-Learning. The plots for the rewards are taken with a moving average across a window of 10 episodes. This smoothens the irregularities of the plot enough to be able to make good observations about the general trends.

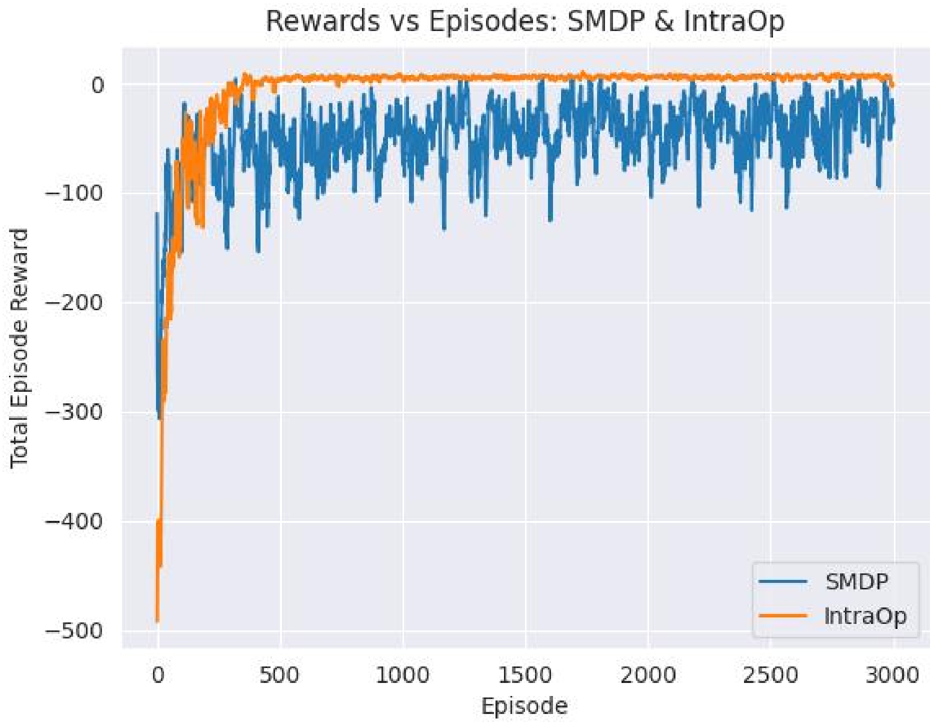


Figure 22: Reward Curve: SMDP vs Intra - Option Q-Learning

As we can clearly see, Intra-Option Q-Learning performs significantly better than even the best tuned version of SMDP Q-learning. This can be attributed to the fact that intra-option Q-learning makes a number of intra-option updates. The q-values of both primitive actions and options are updated at every step. This allows greater flexibility in learning as well as a faster convergence. Another factor is that Intra-Option Q-Learning can learn sub-policies with different durations, allowing for greater flexibility in the decision-making process.

Now we can compare the tables of update frequencies.

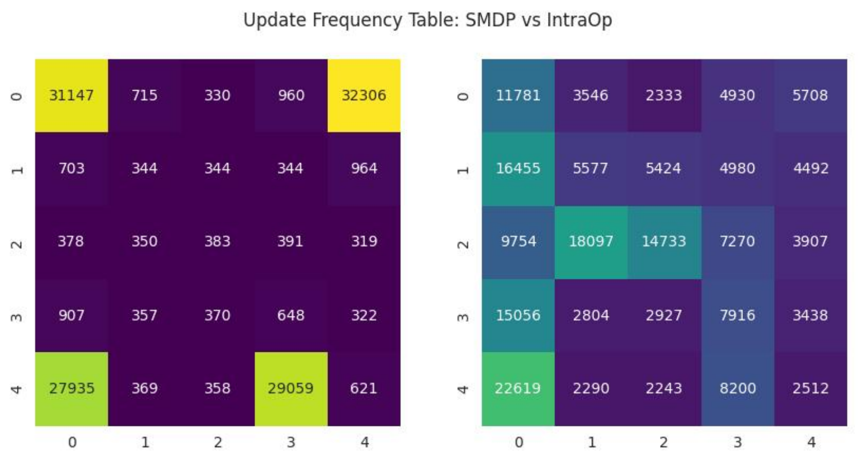


Figure 23: Update Frequency: SMDP vs Intra - Option Q-Learning

As we can clearly see there is a heavy tendency for options to be directly selected in case of SMDP Q-Learning. This is because the option update equations do not change the Q-values of the primitive actions which causes them to be neglected as a result. The option definition is thus very important for SMDP.

7 Alternate Options

7.1 Defining Alternate Options

We have been asked to define an alternate set of options mutually exclusive of the given set of options and compare the performance of both sets. Towards this, we define the options as described below:

Keep South

The policy in this case is to **keep moving south** till termination is reached.

Termination/Initiation set:

- All cells in row 4

Note that this definition is possible as we do not have any horizontal walls to block southward movement.

Keep North

The policy in this case is to **keep moving north** till termination is reached.

Termination/Initiation set:

- All cells in row 0

This definition is possible as there are no horizontal walls to block northward movement.

Keep East

The policy in this case is to **keep moving east** till termination is reached.

Termination/Initiation set:

- All cells in column 4
- Cells (0, 1), (1, 1), (3, 0), (3, 2), (4, 0), (4, 2)

Note that the set is more involved here due to the presence of vertical walls.

Keep West

The policy in this case is to **keep moving west** till termination is reached.

Termination/Initiation set:

- All cells in column 0
- Cells (0, 2), (1, 2), (3, 1), (3, 3), (4, 1), (4, 3)

7.2 Comparing Results for Intra Option Q-Learning

We carry out hyperparameter tuning on the hyperparameters like α (learning rate), and ϵ using **average reward**. We test the following combinations of hyperparameters (total 12 configurations): α: [0.5, 0.1, 0.05, 0.01] and ϵ: [0.1, 0.01, 0.001]

The average reward curve obtained on the best hyperparameters by averaging rewards over 10 runs is plotted below:

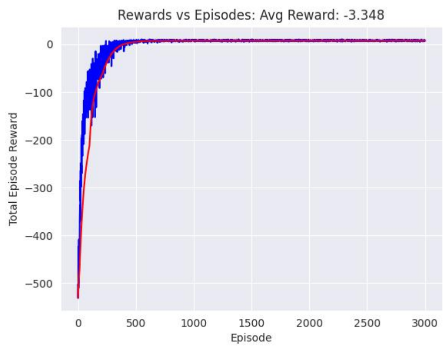


Figure 24: IntraOption Reward Curve: New Options

We can also compare the plots for the old and new rewards as given below.

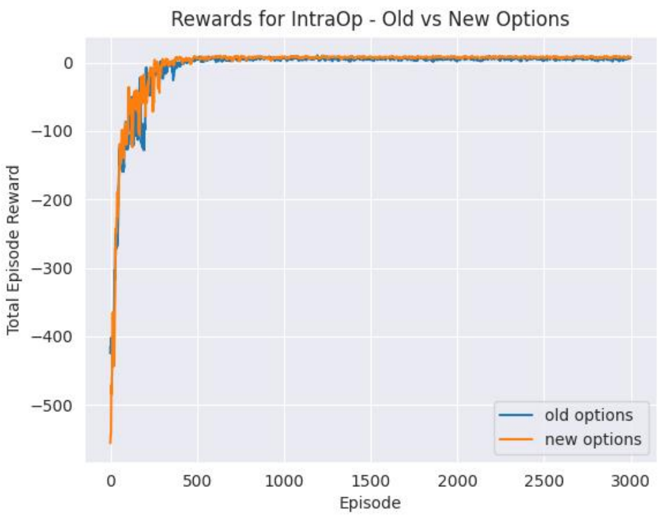


Figure 25: Intra Option Rewards - Old vs New Options

If we notice it carefully, we can see that the new options seem to be performing slightly better.

7.3 Comparing Results for SMDP Q-Learning

We carry out hyperparameter tuning on the hyperparameter α (learning rate) We test the following combinations of hyperparameters (total 4 configurations): α: [0.5, 0.1, 0.05, 0.01]

The average reward curve obtained on the best hyperparameters by averaging rewards over 10 runs is plotted below:

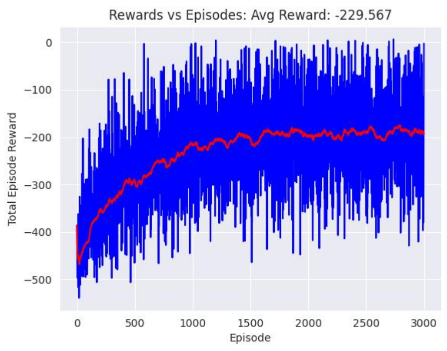


Figure 26: SMDP Reward Curve: New Options

We can also compare the plots for the old and new rewards as given below.

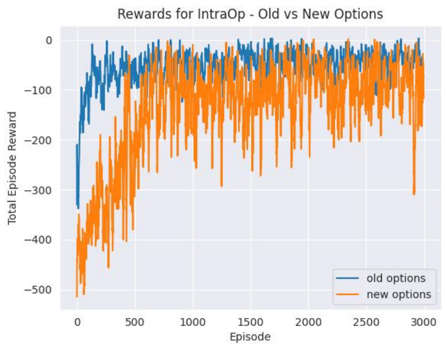


Figure 27: SMDP Rewards - Old vs New Options

As we can see SMDP performs **very poorly** for the newly defined options. This can be attributed to the fact that most of the value updates take place for options in SMDP. The new options being ill-defined for the problem at hand thus greatly affects the performance of SMDP almost completely preventing good convergence.

8 Conclusion

In this assignment, we have looked at two popular option-based methods: SMDP q-learning and intra-option q-learning. We extensively analyzed their performance on the 'Taxi-v3' environment, empirically arriving at the following conclusions:

- Both SMDP and Intra-Option q-learning improve over classical q-learning making use of options.
- Empirically, intra-option outperforms SMDP q-learning. This could be attributed to the higher update frequency.
- Due to the lower update-frequency of primitive actions, SMDP relies heavily on the availability of precise and goal-oriented options as can be seen from the effect of using less precise options.
- Intra-option q-learning is relatively more robust to the quality of options as can be seen from the fact that the performance practically remains unchanged even when lower quality options are used.