CS6700_Tutorial_4_QLearning_SARSA_ME19B190

February 19, 2023

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from IPython.display import clear_output
%matplotlib inline
```

1 Problem Statement

In this section we will implement tabular SARSA and Q-learning algorithms for a grid world navigation task.

1.1 Environment details

The agent can move from one grid coordinate to one of its adjacent grids using one of the four actions: UP, DOWN, LEFT and RIGHT. The goal is to go from a randomly assigned starting position to goal position.

Actions that can result in taking the agent off the grid will not yield any effect. Lets look at the environment.

[2]:
```
DOWN = 0
UP = 1
LEFT = 2
RIGHT = 3
actions = [DOWN, UP, LEFT, RIGHT]
```

Let us construct a grid in a text file.

[3]:
```
!cat grid_world2.txt
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
```

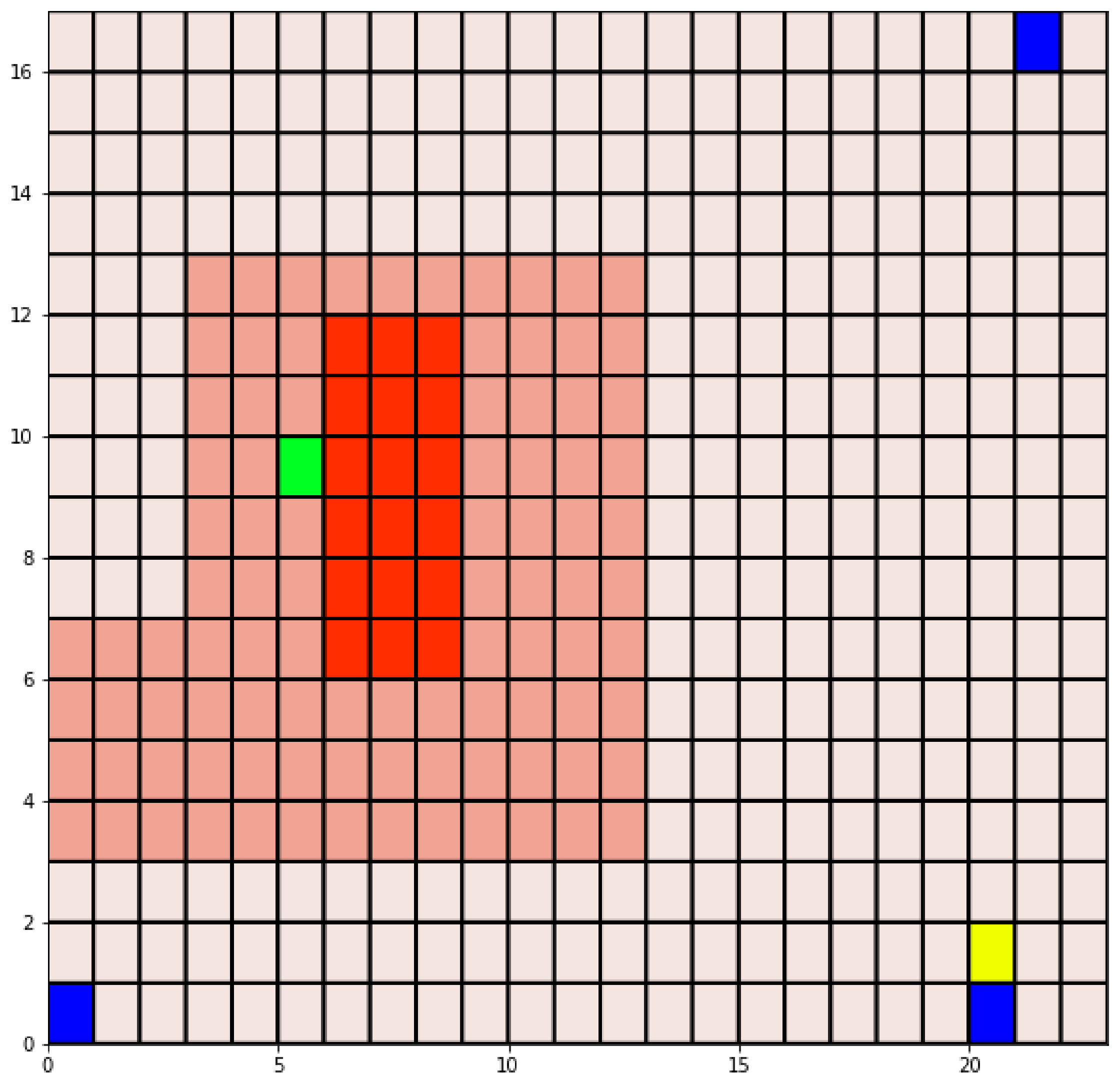
```
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

This is a 17×23 grid. The reward when an agent goes to a cell is negative of the value in that position in the text file (except if it is the goal cell). We will define the goal reward as 100. We will also fix the maximum episode length to 10000.

Now let's make it more difficult. We add stochasticity to the environment: with probability 0.2 agent takes a random action (which can be other than the chosen action). There is also a westerly wind blowing (to the right). Hence, after every time-step, with probability 0.5 the agent also moves an extra step to the right.

Now let's plot the grid world.

[4]:
```
world = 'grid_world2.txt'
goal_reward = 100
start_states = [(0,0), (0,20), (16,21)]
goal_states=[(9,5)]
max_steps=10000

from grid_world import GridWorldEnv, GridWorldWindyEnv

env = GridWorldEnv(world, goal_reward=goal_reward, start_states=start_states,␣
↪goal_states=goal_states,
                max_steps=max_steps, action_fail_prob=0.2)
plt.figure(figsize=(10, 10))
# Go UP
env.step(UP)
env.render(ax=plt, render_agent=True)
```



1.1.1 Legend

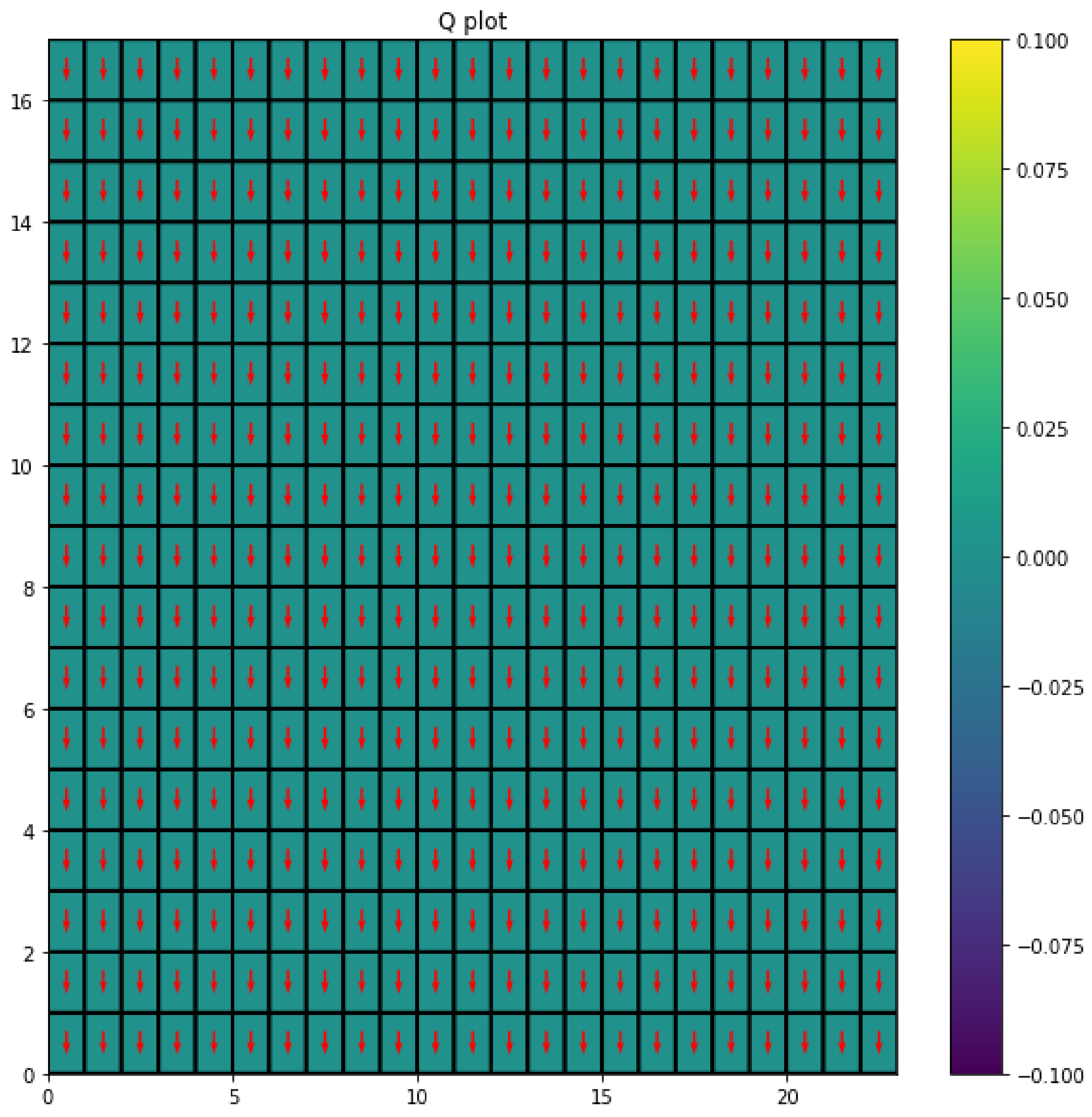
- *Blue* is the **start state**.
- *Green* is the **goal state**.
- *Yellow* is current **state of the agent**.
- *Redness* denotes the extent of **negative reward**.

1.1.2 Q values

We can use a 3D array to represent Q values. The first two indices are X, Y coordinates and last index is the action.

[6]:
```python
from grid_world import plot_Q

Q = np.zeros((env.grid.shape[0], env.grid.shape[1], len(env.action_space)))
```

```
plot_Q(Q)

Q.shape
```



```
[6]: (17, 23, 4)
```

1.1.3 Exploration strategies

1. Epsilon-greedy
2. Softmax

[7]:
```
from scipy.special import softmax

seed = 42
rg = np.random.RandomState(seed)

# Epsilon greedy
def choose_action_epsilon(Q, state, epsilon, rg=rg):

    if (not Q[state[0], state[1]].any()) or (np.random.uniform(0,1) < epsilon):␣
 ↪# TODO: eps greedy condition
        return (np.random.choice(np.arange(len(Q[state[0], state[1]])),1)[0]) #␣
 ↪TODO: return random action
    else:
        max_q_index = np.argmax(Q[state[0], state[1]])
        return max_q_index # TODO: return best action

# Softmax
def choose_action_softmax(Q, state, rg=rg):
    prob = softmax(Q[state[0], state[1]])
    return_action_softmax = np.random.choice(np.arange(len(Q[state[0],␣
 ↪state[1]])),1,p = prob)[0]
    return return_action_softmax # TODO: return random action with selection␣
 ↪probability
```

1.2 SARSA

Now we implement the SARSA algorithm.

Recall the update rule for SARSA:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{1}$$

1.2.1 Hyperparameters

So we have som hyperparameters for the algorithm: - α - number of *episodes*. - ϵ: For epsilon greedy exploration

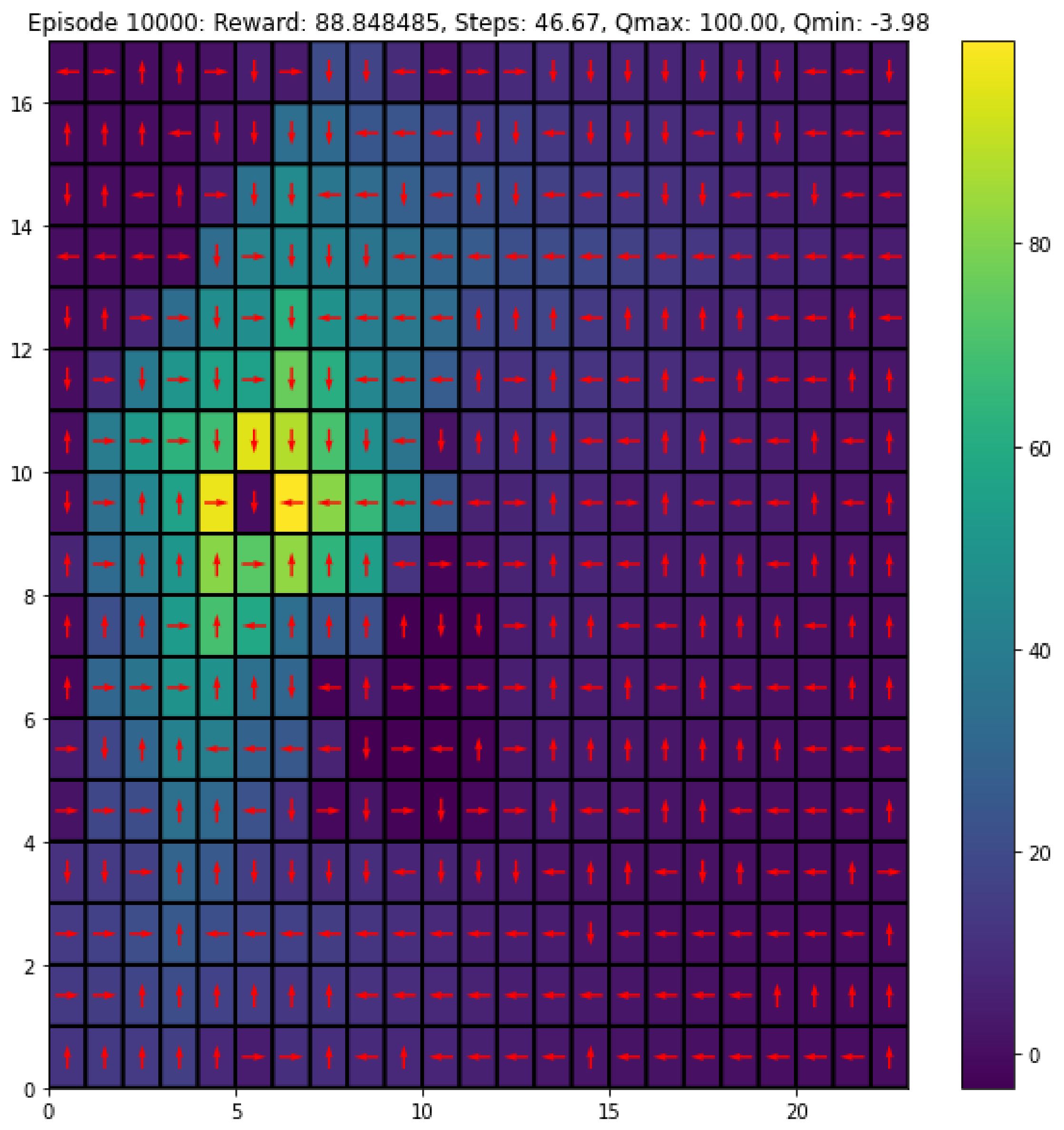
[8]:
```
# initialize Q-value
Q = np.zeros((env.grid.shape[0], env.grid.shape[1], len(env.action_space)))

alpha0 = 0.4
gamma = 0.9
episodes = 10000
epsilon0 = 0.1
```

Let's implement SARSA

[9]:
```
print_freq = 100
```

```
def sarsa(env, Q, gamma = 0.9, plot_heat = False, choose_action =␣
↪choose_action_softmax):

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)
    if plot_heat:
        clear_output(wait=True)
        plot_Q(Q)
    epsilon = epsilon0
    alpha = alpha0
    for ep in tqdm(range(episodes)):
        tot_reward, steps = 0, 0

        # Reset environment
        state = env.reset()
        action = choose_action(Q, state)
        done = False
        while not done:
            state_next, reward, done = env.step(action)
            action_next = choose_action(Q, state_next)

            # TODO: update equation
            Q[state[0],state[1],action]  = Q[state[0],state[1],action] +␣
↪(alpha*(reward + (gamma*(Q[state_next[0],state_next[1],action_next])) - ␣
↪(Q[state[0],state[1],action])))

            tot_reward += reward
            steps += 1

            state, action = state_next, action_next

        episode_rewards[ep] = tot_reward
        steps_to_completion[ep] = steps

        if (ep+1)%print_freq == 0 and plot_heat:
            clear_output(wait=True)
            plot_Q(Q, message = "Episode %d: Reward: %f, Steps: %.2f, Qmax: %.
↪2f, Qmin: %.2f"%(ep+1, np.mean(episode_rewards[ep-print_freq+1:ep]),
                                                                          np.
↪mean(steps_to_completion[ep-print_freq+1:ep]),
                                                                          Q.
↪max(), Q.min()))

    return Q, episode_rewards, steps_to_completion
```

[10]:
```
Q, rewards, steps = sarsa(env, Q, gamma = gamma, plot_heat=True, choose_action=␣
↪choose_action_softmax)
```



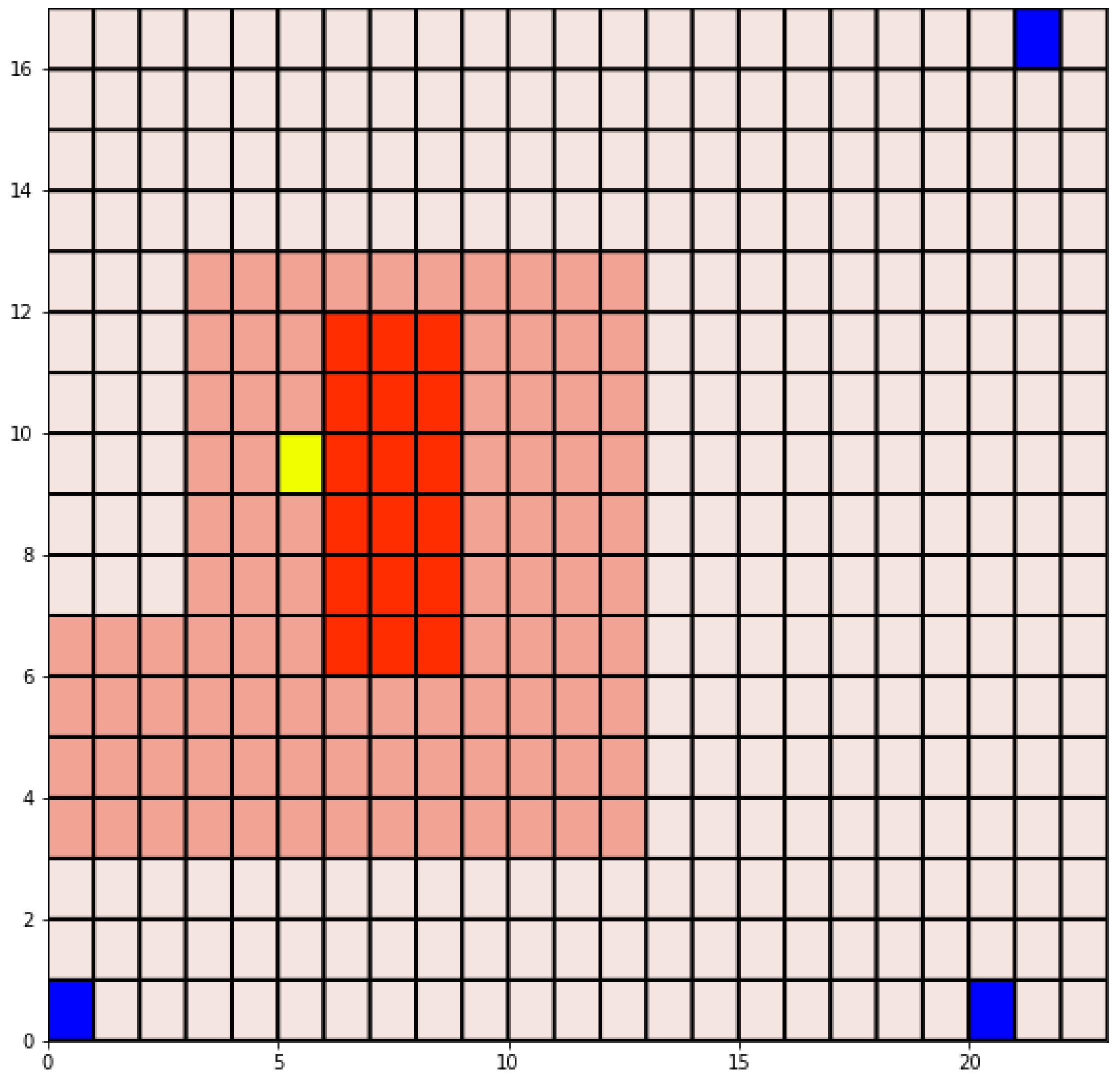
```
100%|     | 10000/10000 [01:59<00:00, 83.66it/s]
```

1.2.2 Visualizing the policy

Now let's see the agent in action. Run the below cell (as many times) to render the policy;

[11]:
```
from time import sleep

state = env.reset()
done = False
steps = 0
tot_reward = 0
```

```python
while not done:
    clear_output(wait=True)
    state, reward, done = env.step(Q[state[0], state[1]].argmax())
    plt.figure(figsize=(10, 10))
    env.render(ax=plt, render_agent=True)
    plt.show()
    steps += 1
    tot_reward += reward
    sleep(0.2)
print("Steps: %d, Total Reward: %d"%(steps, tot_reward))
```



```
Steps: 22, Total Reward: 88
```

1.2.3 Analyzing performance of the policy

We use two metrics to analyze the policies:

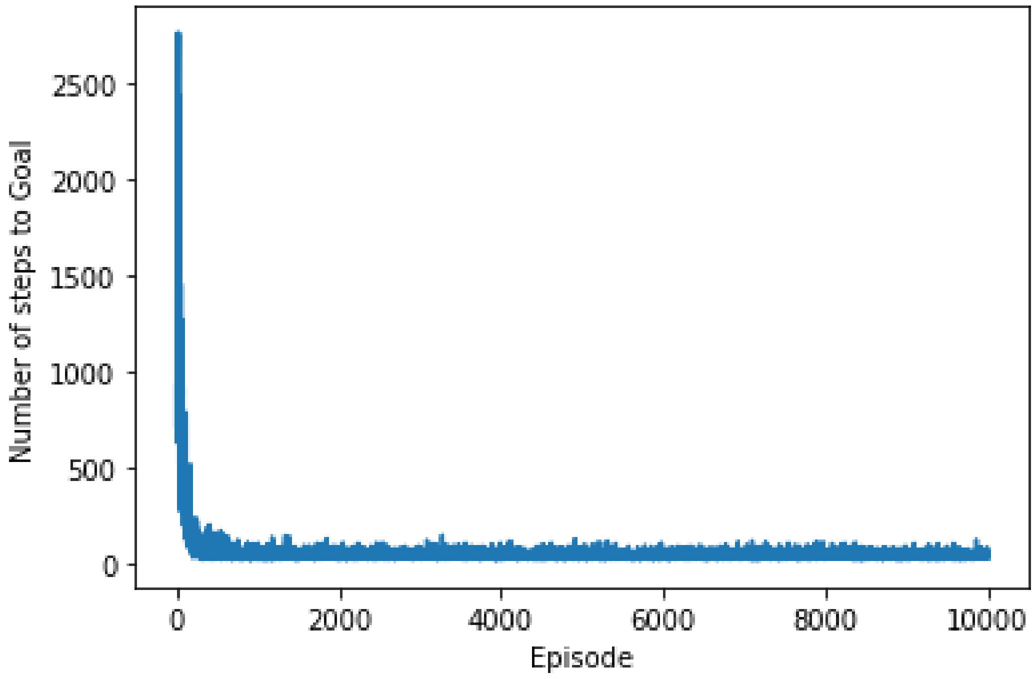
1. Average steps to reach the goal
2. Total rewards from the episode

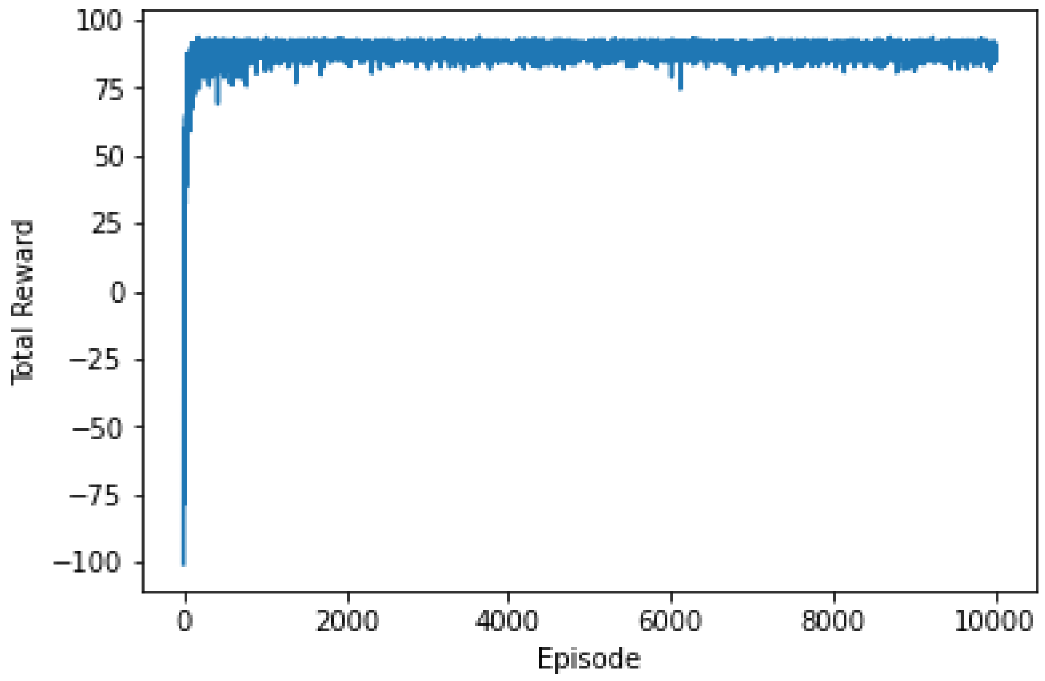
To ensure, we account for randomness in environment and algorithm (say when using epsilon-greedy exploration), we run the algorithm for multiple times and use the average of values over all runs.

[12]:
```
num_expts = 5
reward_avgs, steps_avgs = [], []

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((env.grid.shape[0], env.grid.shape[1], len(env.action_space)))
    rg = np.random.RandomState(i)
    Q, rewards, steps = sarsa(env, Q, gamma = gamma, plot_heat=False,␣
↪choose_action= choose_action_softmax)
    reward_avgs.append(rewards)
    steps_avgs.append(steps)

reward_avgs = np.array(reward_avgs).mean(axis = 0)
steps_avgs  = np.array(steps_avgs).mean(axis = 0)

    # TODO: run sarsa, store metrics
```

```
Experiment: 1

100%|     | 10000/10000 [01:19<00:00, 125.05it/s]

Experiment: 2

100%|     | 10000/10000 [00:55<00:00, 179.63it/s]

Experiment: 3

100%|     | 10000/10000 [01:18<00:00, 127.86it/s]

Experiment: 4

100%|     | 10000/10000 [01:30<00:00, 110.21it/s]

Experiment: 5

100%|     | 10000/10000 [00:49<00:00, 202.37it/s]
```

[13]:
```
# TODO: visualize individual metrics vs episode count (averaged across multiple␣
↪run(s))

plt.figure()
plt.plot( np.arange(episodes),steps_avgs)
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.show()

plt.figure()
plt.plot(np.arange(episodes),reward_avgs)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```





1.3 Q-Learning

Now, implement the Q-Learning algorithm as an exercise.

Recall the update rule for Q-Learning:

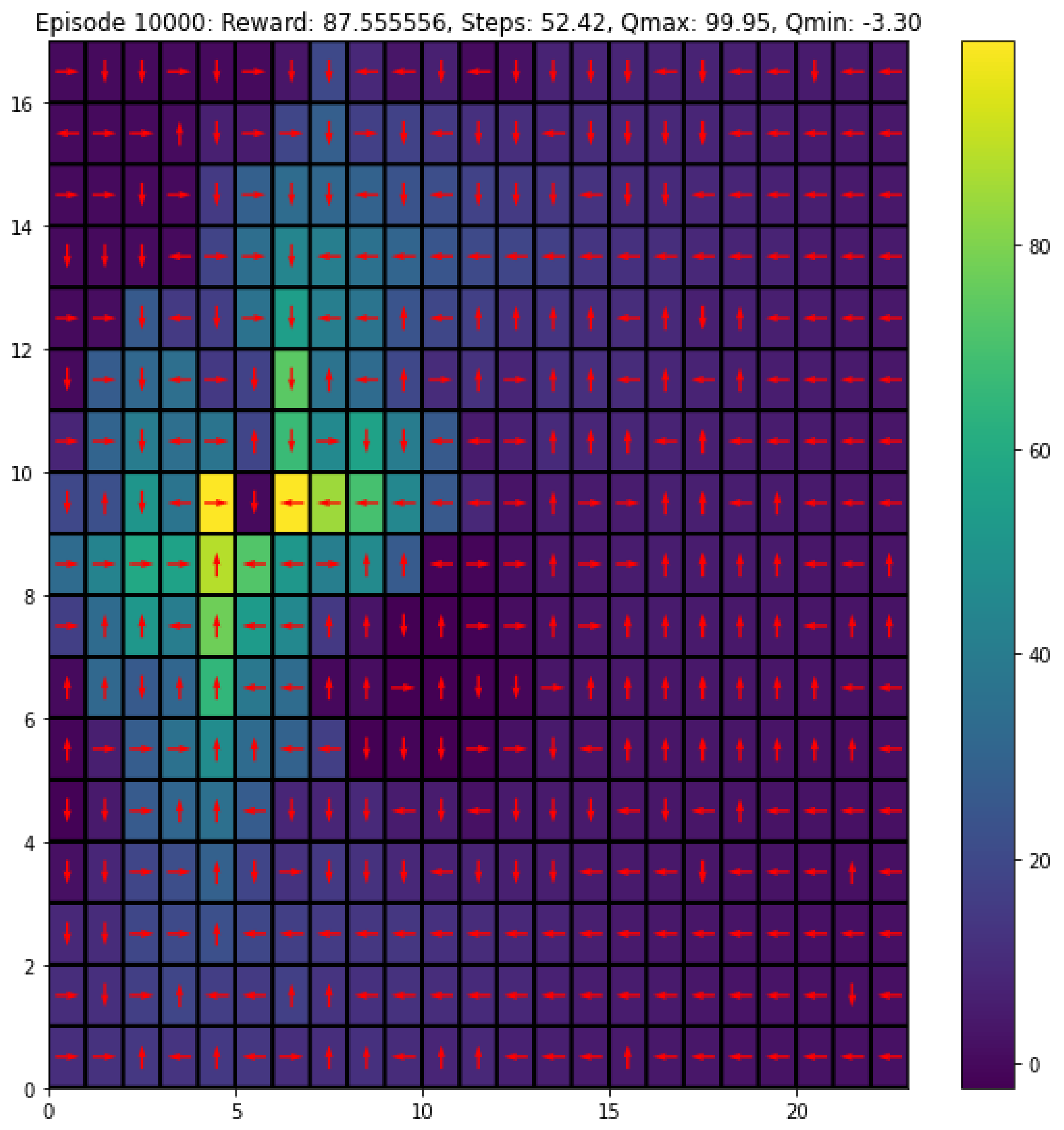
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2}$$

Visualize and compare results with SARSA.

[14]:
```
# initialize Q-value
Q = np.zeros((env.grid.shape[0], env.grid.shape[1], len(env.action_space)))

alpha0 = 0.4
gamma = 0.9
episodes = 10000
epsilon0 = 0.1
```

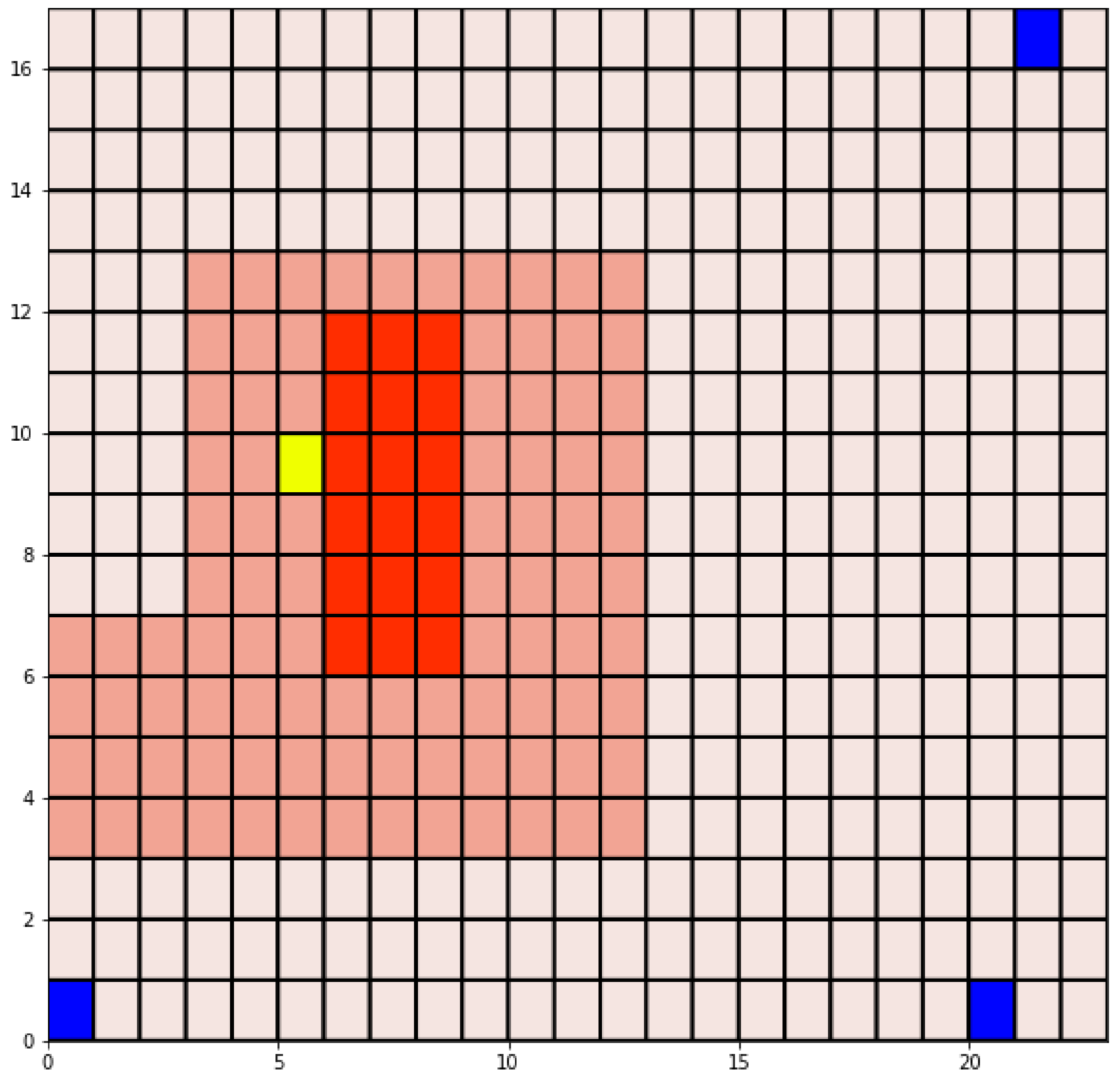
[15]:
```
print_freq = 100

def qlearning(env, Q, gamma = 0.9, plot_heat = False, choose_action =␣
↪choose_action_softmax):

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)
    if plot_heat:
        clear_output(wait=True)
        plot_Q(Q)
    epsilon = epsilon0
    alpha = alpha0
    for ep in tqdm(range(episodes)):
        tot_reward, steps = 0, 0

        # Reset environment
        state = env.reset()
        action = choose_action(Q, state)
        done = False
        while not done:
            state_next, reward, done = env.step(action)
            action_next = choose_action(Q, state_next)

            # TODO: update equation
            Q[state[0],state[1],action]  = Q[state[0],state[1],action] +␣
↪(alpha*(reward+(gamma*max(Q[state_next[0],state_next[1]]))-␣
↪Q[state[0],state[1],action]))

            tot_reward += reward
            steps += 1

            state, action = state_next, action_next

        episode_rewards[ep] = tot_reward
        steps_to_completion[ep] = steps

        if (ep+1)%print_freq == 0 and plot_heat:
            clear_output(wait=True)
            plot_Q(Q, message = "Episode %d: Reward: %f, Steps: %.2f, Qmax: %.
↪2f, Qmin: %.2f"%(ep+1, np.mean(episode_rewards[ep-print_freq+1:ep]),
                                                                          np.
↪mean(steps_to_completion[ep-print_freq+1:ep]),
                                                                          Q.
↪max(), Q.min()))

    return Q, episode_rewards, steps_to_completion
```

[16]:
```
Q, rewards, steps = qlearning(env, Q, gamma = gamma, plot_heat=True,␣
↪choose_action= choose_action_softmax)
```



```
100%|          | 10000/10000 [01:46<00:00, 94.29it/s]
```

[17]:
```
from time import sleep

state = env.reset()
done = False
steps = 0
tot_reward = 0
while not done:
    clear_output(wait=True)
    state, reward, done = env.step(Q[state[0], state[1]].argmax())
    plt.figure(figsize=(10, 10))
    env.render(ax=plt, render_agent=True)
    plt.show()
    steps += 1
    tot_reward += reward
    sleep(0.2)
print("Steps: %d, Total Reward: %d"%(steps, tot_reward))
```

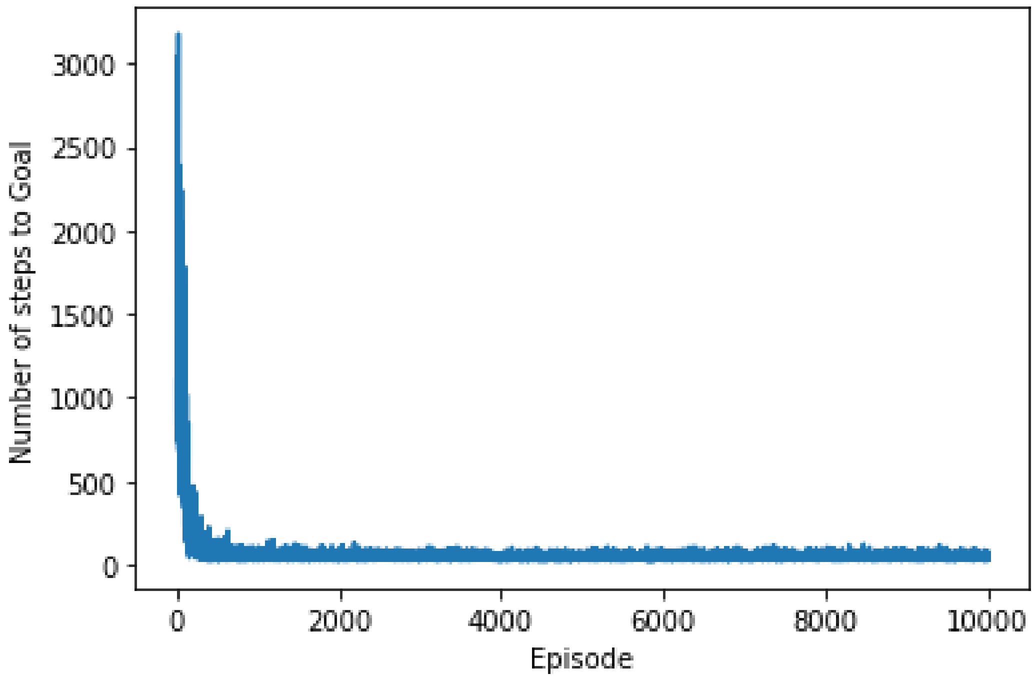


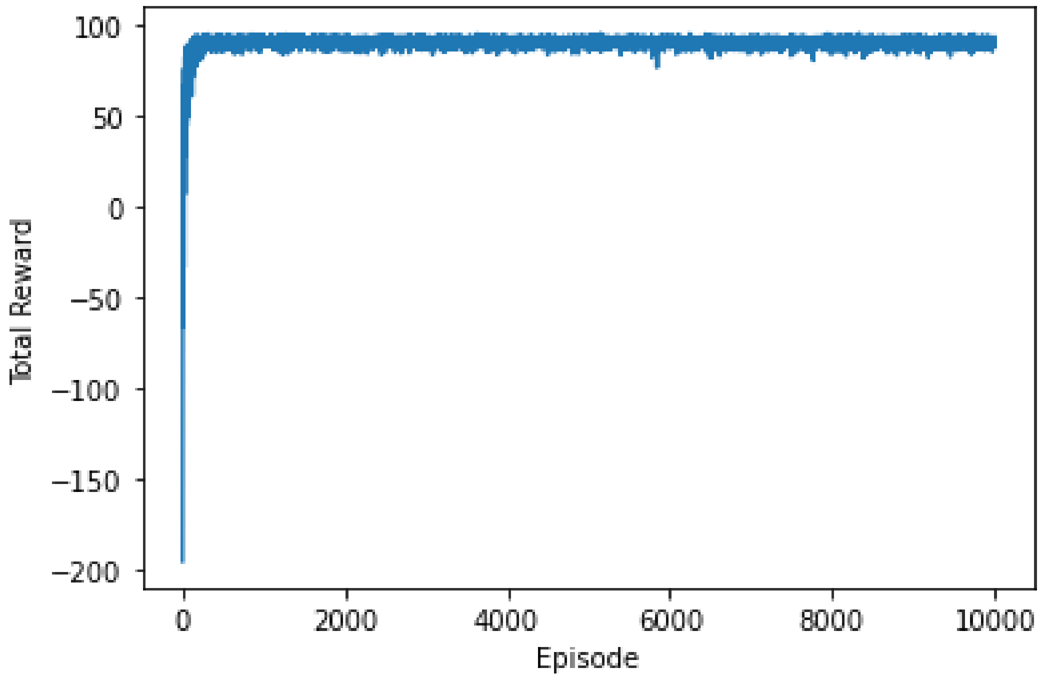
```
Steps: 20, Total Reward: 91
```

[18]:
```
num_expts = 5
reward_avgs, steps_avgs = [], []

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((env.grid.shape[0], env.grid.shape[1], len(env.action_space)))
    rg = np.random.RandomState(i)
    Q, rewards, steps = sarsa(env, Q, gamma = gamma, plot_heat=False,␣
 ↪choose_action= choose_action_softmax)
    reward_avgs.append(rewards)
    steps_avgs.append(steps)

reward_avgs = np.array(reward_avgs).mean(axis = 0)
steps_avgs  = np.array(steps_avgs).mean(axis = 0)

    # TODO: run qlearning, store metrics
```

```
Experiment: 1

100%|      | 10000/10000 [01:21<00:00, 122.64it/s]

Experiment: 2

100%|      | 10000/10000 [01:35<00:00, 104.99it/s]

Experiment: 3

100%|      | 10000/10000 [01:10<00:00, 141.79it/s]

Experiment: 4

100%|      | 10000/10000 [01:53<00:00, 87.73it/s]

Experiment: 5

100%|      | 10000/10000 [01:19<00:00, 126.00it/s]
```

[19]:
```
# TODO: visualize individual metrics vs episode count (averaged across multiple␣
 ↪run(s))

plt.figure()
plt.plot( np.arange(episodes),steps_avgs)
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.show()

plt.figure()
plt.plot(np.arange(episodes),reward_avgs)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```


Number of steps to Goal
3000
2500
2000
1500
1000
500
0
0
2000
4000
6000
8000
10000
Episode


Total Reward
100
50
0
−50
−100
−150
−200
0
2000
4000
6000
8000
10000
Episode

1.3.1 TODO: What differences do you observe between the policies learnt by Q Learning and SARSA (if any).

Q learning: An action is taken using an exploration strategy(epsilon greedy or softmax) and the Q value is updated using the maximum value of action value function of the next state action pair.

Sarsa: An action is taken using an exploration strategy(epsilon greedy or softmax) and the Q value is updated using the value of action value function of the next state action pair, where the next action is selected using one of the exploration strategies.

SARSA will try to maximize the mean reward and try to return the safest path possible(might not be the shortest), where as Q-learning is greedy with respect to each step and thus, tries to return the shortest path possible. Q learning will have less mean reward compared to SARSA over time.

Since, the environment has stochasticity, it is better to implement SARSA compared to Q learning as it returns a more safer path. Although Q learning has a better convergence rate compared to SARSA, the advantages of implementing one over other depends on the stochasticity involed in the environment.

[21]:
```
!pip install nbconvert
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: nbconvert in /usr/local/lib/python3.8/dist-
packages (5.6.1)
Requirement already satisfied: pandocfilters>=1.4.1 in
/usr/local/lib/python3.8/dist-packages (from nbconvert) (1.5.0)
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.8/dist-
packages (from nbconvert) (5.2.0)
Requirement already satisfied: testpath in /usr/local/lib/python3.8/dist-
packages (from nbconvert) (0.6.0)
Requirement already satisfied: defusedxml in /usr/local/lib/python3.8/dist-
packages (from nbconvert) (0.7.1)
Requirement already satisfied: nbformat>=4.4 in /usr/local/lib/python3.8/dist-
packages (from nbconvert) (5.7.3)
Requirement already satisfied: entrypoints>=0.2.2 in
/usr/local/lib/python3.8/dist-packages (from nbconvert) (0.4)
Requirement already satisfied: pygments in /usr/local/lib/python3.8/dist-
packages (from nbconvert) (2.6.1)
Requirement already satisfied: jinja2>=2.4 in /usr/local/lib/python3.8/dist-
packages (from nbconvert) (2.11.3)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.8/dist-
packages (from nbconvert) (5.7.1)
Requirement already satisfied: mistune<2,>=0.8.1 in
/usr/local/lib/python3.8/dist-packages (from nbconvert) (0.8.4)
Requirement already satisfied: bleach in /usr/local/lib/python3.8/dist-packages
(from nbconvert) (6.0.0)
Requirement already satisfied: MarkupSafe>=0.23 in
```

```
/usr/local/lib/python3.8/dist-packages (from jinja2>=2.4->nbconvert) (2.0.1)
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.8/dist-
packages (from nbformat>=4.4->nbconvert) (4.3.3)
Requirement already satisfied: fastjsonschema in /usr/local/lib/python3.8/dist-
packages (from nbformat>=4.4->nbconvert) (2.16.2)
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.8/dist-
packages (from bleach->nbconvert) (1.15.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.8/dist-
packages (from bleach->nbconvert) (0.5.1)
Requirement already satisfied: platformdirs>=2.5 in
/usr/local/lib/python3.8/dist-packages (from jupyter-core->nbconvert) (3.0.0)
Requirement already satisfied: attrs>=17.4.0 in /usr/local/lib/python3.8/dist-
packages (from jsonschema>=2.6->nbformat>=4.4->nbconvert) (22.2.0)
Requirement already satisfied: importlib-resources>=1.4.0 in
/usr/local/lib/python3.8/dist-packages (from
jsonschema>=2.6->nbformat>=4.4->nbconvert) (5.10.2)
Requirement already satisfied: pyrsistent!=0.17.0,!=0.17.1,!=0.17.2,>=0.14.0 in
/usr/local/lib/python3.8/dist-packages (from
jsonschema>=2.6->nbformat>=4.4->nbconvert) (0.19.3)
Requirement already satisfied: zipp>=3.1.0 in /usr/local/lib/python3.8/dist-
packages (from importlib-
resources>=1.4.0->jsonschema>=2.6->nbformat>=4.4->nbconvert) (3.13.0)
Reading package lists… Done
Building dependency tree
Reading state information… Done
texlive-fonts-recommended is already the newest version (2019.20200218-1).
texlive-plain-generic is already the newest version (2019.202000218-1).
texlive-xetex is already the newest version (2019.20200218-1).
The following package was automatically installed and is no longer required:
  libnvidia-common-510
Use 'sudo apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 21 not upgraded.
```

[36]:
```
!jupyter nbconvert --to pdf "/content/
↪CS6700_Tutorial_4_QLearning_SARSA_ME19B190.ipynb"
```

```
[NbConvertApp] Converting notebook
/content/CS6700_Tutorial_4_QLearning_SARSA_ME19B190.ipynb to pdf
[NbConvertApp] Support files will be in
CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files/
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
```

```
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
[NbConvertApp] Making directory
./CS6700_Tutorial_4_QLearning_SARSA_ME19B190_files
[NbConvertApp] Writing 74162 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', './notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', './notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 383730 bytes to
/content/CS6700_Tutorial_4_QLearning_SARSA_ME19B190.pdf
```