```
import gym
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import softmax
from IPython.display import clear_output
from time import sleep
plt.style.use("ggplot")

# import the gym environment
env = gym.make("Taxi-v3")
```

```
C:\Users\Varun Gumma\AppData\Local\Programs\Python\Python310\lib\site-
packages\gym\wrappers\monitoring\video_recorder.py:9:
DeprecationWarning: The distutils package is deprecated and slated for
removal in Python 3.12. Use setuptools or check PEP 632 for potential
alternatives
  import distutils.spawn
```

Options

```
# primitive actions
N, S, E, W, P, D = 1, 0, 2, 3, 4, 5

# options
Opt_Reach_R, Opt_Reach_G, Opt_Reach_B, Opt_Reach_Y = 6, 7, 8, 9

# co-ordinates of R,G,Y,B
R, G, Y, B = [0, 0], [0, 4], [4, 0], [4, 3]

# step-wise actions to lead to R/G/Y/B
towards_R = [[N,W,S,S,S],
             [N,W,S,S,S],
             [N,W,W,W,W],
             [N,N,N,N,N],
             [N,N,N,N,N]]

towards_G = [[S,S,E,E,E],
             [S,S,E,E,N],
             [E,E,E,E,N],
             [N,N,N,N,N],
             [N,N,N,N,N]]

towards_Y = [[S,S,S,S,S],
             [S,S,S,S,S],
             [S,W,W,W,W],
             [S,N,N,N,N],
             [W,N,N,N,N]]

towards_B =  [[S,S,S,S,S],
```

```
                [S,S,S,S,S],
                [E,E,E,S,W],
                [N,N,N,S,W],
                [N,N,N,S,W]]

q_values_SMDP = np.zeros((500,  10))
q_values_IO = np.zeros((500,  10))

def egreedy_policy(q_values, state, epsilon):
    if np.random.random() > epsilon:
        return q_values[state].argmax()
    else:
        return np.random.choice(q_values.shape[1])

def softmax_policy(q_values, state, beta):
    p = softmax(q_values[state]/beta)
    return np.random.choice(q_values.shape[1], p=p)

def Reach_R(r, c, p_loc, dest):
    optact = towards_R[r][c]
    optdone = False
    if [r, c] == R:
        if p_loc == 0:
            optact = P
        elif p_loc == 4 and dest == 0:
            optact = D
        optdone = True
    return [optact, optdone]

def Reach_B(r, c, p_loc, dest):
    optact = towards_B[r][c]
    optdone = False
    if [r, c] == B:
        if p_loc == 3:
            optact = P
        elif p_loc == 4 and dest == 3:
            optact = D
        optdone = True
    return [optact, optdone]

def Reach_G(r, c, p_loc, dest):
    optact = towards_G[r][c]
    optdone = False
    if [r, c] == G:
        if p_loc == 1:
            optact = P
        elif p_loc == 4 and dest == 1:
            optact = D
        optdone = True
    return [optact, optdone]
```

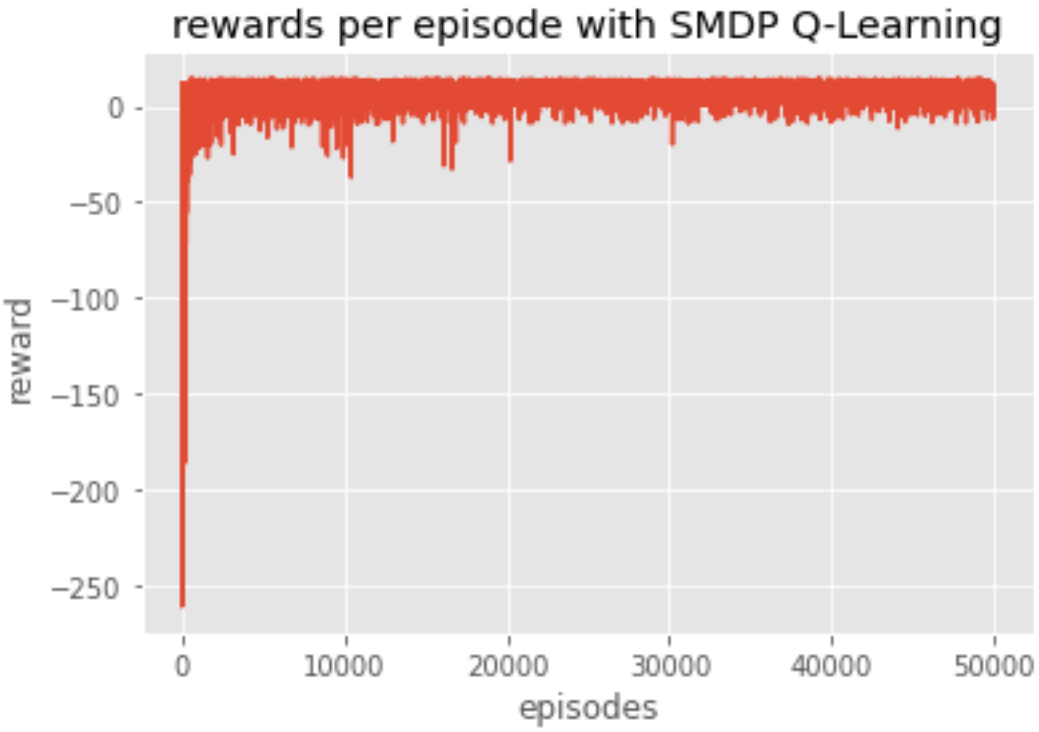
```
def Reach_Y(r, c, p_loc, dest):
    optact = towards_Y[r][c]
    optdone = False
    if [r, c] == Y:
        if p_loc == 2:
            optact = P
        elif p_loc == 4 and dest == 2:
            optact = D
        optdone = True
    return [optact, optdone]
```

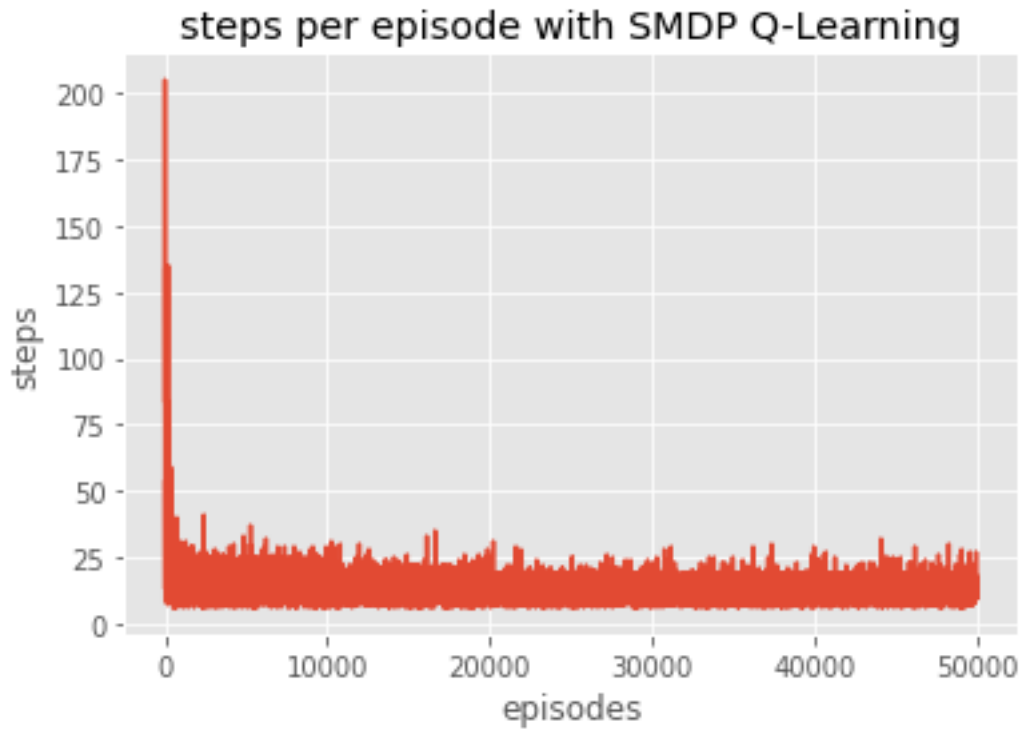
SMDP Q-Learning

```
gamma, lr = 0.99, 0.75
rewards_per_episode_SMDP, steps_per_episode_SMDP = [], []
action_freq_SMDP = [0]*10

for _ in range(50000):
    state = env.reset()
    done = False
    total_reward = 0
    total_steps = 0

    while not done:
        reward_bar, T = 0, 0
        action = egreedy_policy(q_values_SMDP, state, epsilon=0.01)
        action_freq_SMDP[action] += 1

        # primitive action
        if action < 6:
            next_state, reward, done, _ = env.step(action)
            total_reward += reward
            q_values_SMDP[state, action] += (lr * (reward + (gamma *
q_values_SMDP[next_state].max()) - q_values_SMDP[state, action]))
            state = next_state
            total_steps += 1

        # Checking if action chosen is an option
        elif action == Opt_Reach_R: # option to reach R
            optdone = False
            state_t = state
            while not optdone:
                optact, optdone = Reach_R(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                reward_bar  += ((gamma ** T) * reward)
                state = next_state
                total_steps += 1
                T += 1
            total_reward += reward_bar
            q_values_SMDP[state_t, action] += (lr * (reward_bar +
```

```
((gamma ** T) * q_values_SMDP[state].max()) - q_values_SMDP[state_t,
action]))

        elif action == Opt_Reach_G: # option to reach G
            optdone = False
            state_t = state
            while not optdone:
                optact, optdone = Reach_G(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                reward_bar  += ((gamma ** T) * reward)
                state = next_state
                total_steps += 1
                T += 1
            total_reward += reward_bar
            q_values_SMDP[state_t, action] += (lr * (reward_bar +
((gamma ** T) * q_values_SMDP[state].max()) - q_values_SMDP[state_t,
action]))

        elif action == Opt_Reach_B: # option to reach B
            optdone = False
            state_t = state
            while not optdone:
                optact, optdone = Reach_B(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                reward_bar  += ((gamma ** T) * reward)
                state = next_state
                total_steps += 1
                T += 1
            total_reward += reward_bar
            q_values_SMDP[state_t, action] += (lr * (reward_bar +
((gamma ** T) * q_values_SMDP[state].max()) - q_values_SMDP[state_t,
action]))

        elif action == Opt_Reach_Y: # option to reach Y
            optdone = False
            state_t = state
            while not optdone:
                optact, optdone = Reach_Y(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                reward_bar  += ((gamma ** T) * reward)
                state = next_state
                total_steps += 1
                T += 1
            total_reward += reward_bar
            q_values_SMDP[state_t, action] += (lr * (reward_bar +
((gamma ** T) * q_values_SMDP[state].max()) - q_values_SMDP[state_t,
action]))

    rewards_per_episode_SMDP.append(total_reward)
    steps_per_episode_SMDP.append(total_steps)
```

```
plt.plot(rewards_per_episode_SMDP)
plt.xlabel("episodes")
plt.ylabel("reward")
plt.title("rewards per episode with SMDP Q-Learning")
plt.show()

plt.plot(steps_per_episode_SMDP)
plt.xlabel("episodes")
plt.ylabel("steps")
plt.title("steps per episode with SMDP Q-Learning")
plt.show()

plt.bar(range(10), action_freq_SMDP, width=0.3)
plt.xlabel("# action")
plt.ylabel("frequency")
plt.title("Frequency of the actions being chosen with SMDP")
plt.show()

plt.figure(figsize=(15, 100))
plt.pcolor(q_values_SMDP, edgecolors='k', linewidths=0.5,
cmap="viridis")
plt.title("Heat-Map for SMDP Q-Learning")
plt.xticks(np.arange(0, 11, 1.0))
plt.colorbar(shrink=0.1)
plt.show()
```






```
C:\Users\Varun Gumma\AppData\Local\Temp\
ipykernel_5512\1595092481.py:20: MatplotlibDeprecationWarning: Auto-
removal of grids by pcolor() and pcolormesh() is deprecated since 3.5
and will be removed two minor releases later; please call grid(False)
first.
```

```
  plt.pcolor(q_values_SMDP, edgecolors='k', linewidths=0.5,
cmap="viridis")
```

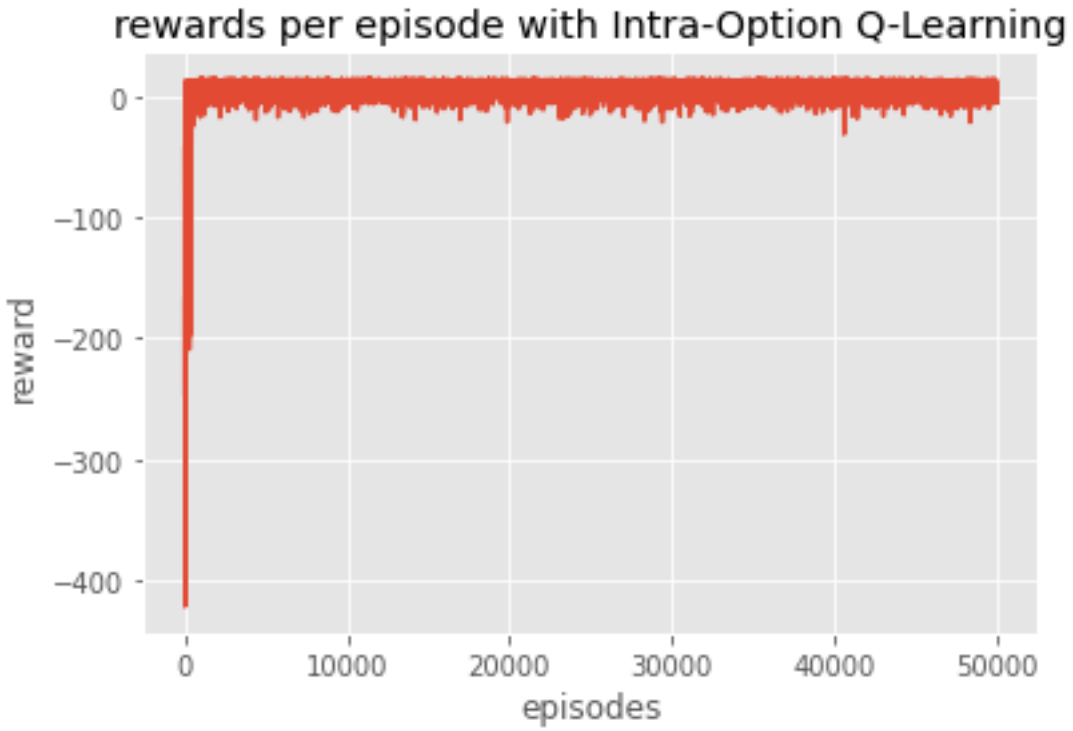

Heat-Map for SMDP Q-Learning
500
400

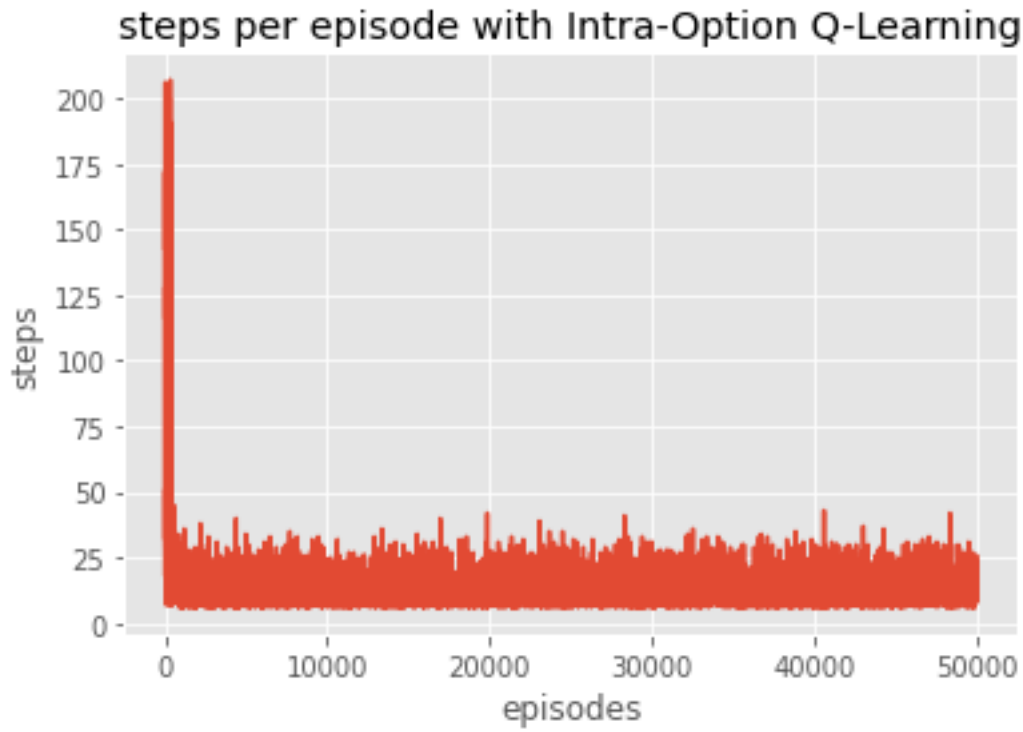
Intra-Option Q-Learning

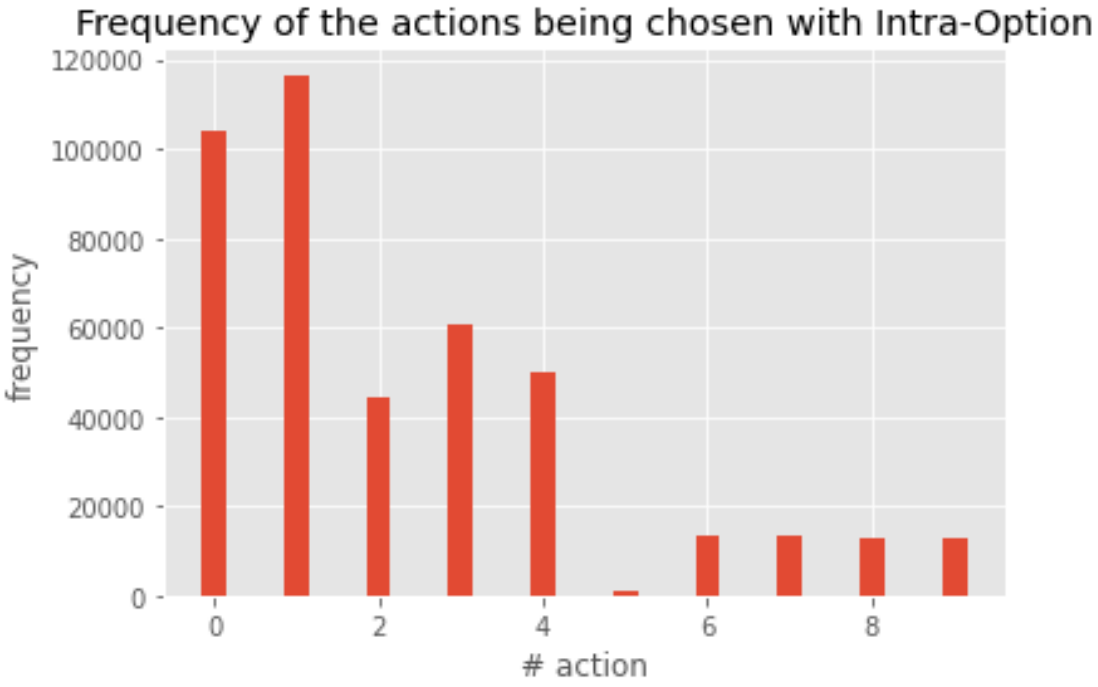
```
def is_drop_off_loc(r, c, dest):
    return (([r, c] == R and dest == 0) or \
            ([r, c] == G and dest == 1) or \
            ([r, c] == Y and dest == 2) or \
            ([r, c] == B and dest == 3))

def is_terminating(next_state):
    r, c, p_loc, dest = env.decode(next_state)
    return p_loc == 4 and is_drop_off_loc(r, c, dest)

gamma, lr = 0.99, 0.75
rewards_per_episode_IO, steps_per_episode_IO = [], []
action_freq_IO = [0]*10

for _ in range(50000):
    state = env.reset()
    done = False
    total_reward = 0
    total_steps = 0


    while not done:
        action = egreedy_policy(q_values_IO, state, epsilon=0.01)
        action_freq_IO[action] += 1

        # Checking if primitive action
        if action < 6:
            next_state, reward, done, _ = env.step(action)
            q_values_IO[state, action] += (lr * (reward + (gamma *
q_values_IO[next_state].max()) - q_values_IO[state, action]))
            state = next_state
            total_reward += reward
            total_steps += 1

        # Checking if action chosen is an option
        elif action == Opt_Reach_R: # option to reach R
            optdone = False
            while not optdone:
                optact, optdone = Reach_R(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                q = q_values_IO[next_state].max() if
is_terminating(next_state) else q_values_IO[next_state, action]
                q_values_IO[state, action] += (lr * (reward + (gamma *
q) - q_values_IO[state, action]))
                state = next_state
                total_reward += reward
                total_steps += 1

        elif action == Opt_Reach_B: # option to reach B
```

```
            optdone = False
            while not optdone:
                optact, optdone = Reach_B(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                q = q_values_IO[next_state].max() if
is_terminating(next_state) else q_values_IO[next_state, action]
                q_values_IO[state, action] += (lr * (reward + (gamma *
q) - q_values_IO[state, action]))
                state = next_state
                total_reward += reward
                total_steps += 1

        elif action == Opt_Reach_G: # option to reach G
            optdone = False
            while not optdone:
                optact, optdone = Reach_G(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                q = q_values_IO[next_state].max() if
is_terminating(next_state) else q_values_IO[next_state, action]
                q_values_IO[state, action] += (lr * (reward + (gamma *
q) - q_values_IO[state, action]))
                state = next_state
                total_reward += reward
                total_steps += 1

        elif action == Opt_Reach_Y: # option to reach Y
            optdone = False
            while not optdone:
                optact, optdone = Reach_Y(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                q = q_values_IO[next_state].max() if
is_terminating(next_state) else q_values_IO[next_state, action]
                q_values_IO[state, action] += (lr * (reward + (gamma *
q) - q_values_IO[state, action]))
                state = next_state
                total_reward += reward
                total_steps += 1

    rewards_per_episode_IO.append(total_reward)
    steps_per_episode_IO.append(total_steps)

plt.plot(rewards_per_episode_IO)
plt.xlabel("episodes")
plt.ylabel("reward")
plt.title("rewards per episode with Intra-Option Q-Learning")
plt.show()

plt.plot(steps_per_episode_IO)
plt.xlabel("episodes")
plt.ylabel("steps")
```

```
plt.title("steps per episode with Intra-Option Q-Learning")
plt.show()

plt.bar(range(10), action_freq_IO, width=0.3)
plt.xlabel("# action")
plt.ylabel("frequency")
plt.title("Frequency of the actions being chosen with Intra-Option")
plt.show()

plt.figure(figsize=(15, 100))
plt.pcolor(q_values_IO, edgecolors='k', linewidths=0.5,
cmap="viridis")
plt.title("Heat-Map for Intra-Option Q-Learning")
plt.xticks(np.arange(0, 11, 1.0))
plt.colorbar(shrink=0.1)
plt.show()
```





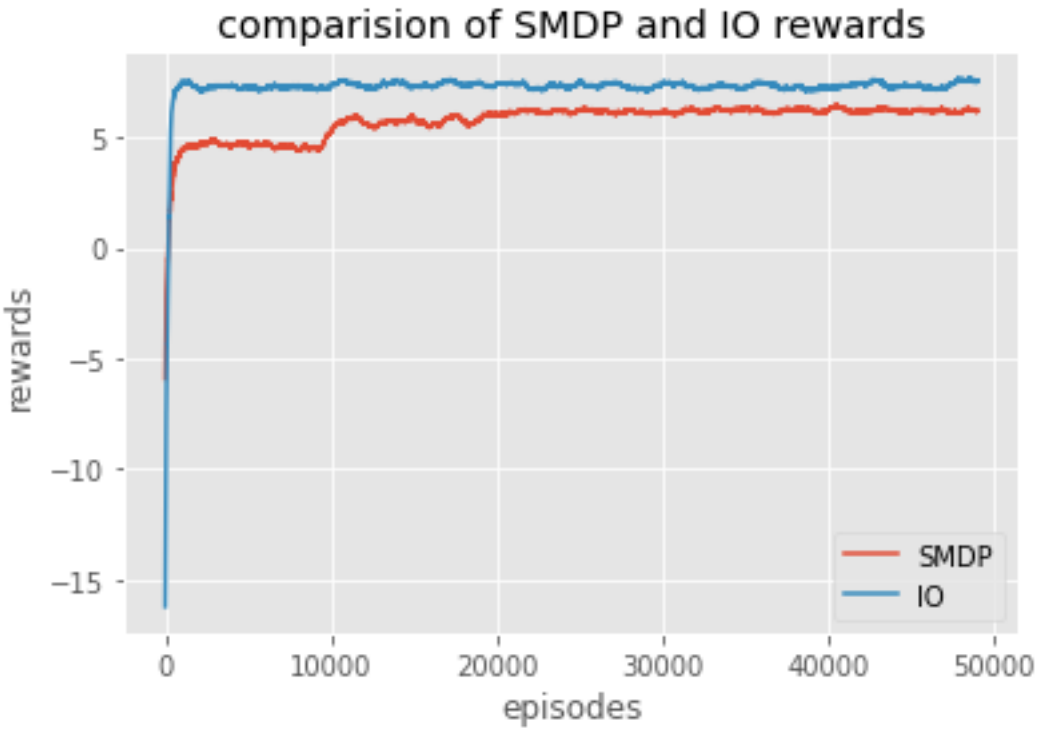



```
C:\Users\Varun Gumma\AppData\Local\Temp\
ipykernel_5512\3989647898.py:20: MatplotlibDeprecationWarning: Auto-
removal of grids by pcolor() and pcolormesh() is deprecated since 3.5
and will be removed two minor releases later; please call grid(False)
first.
```

```
  plt.pcolor(q_values_IO, edgecolors='k', linewidths=0.5,
cmap="viridis")
```


Heat-Map for Intra-Option Q-Learning
500
400

```
W = 1000
plt.plot(np.convolve(rewards_per_episode_SMDP, np.ones(W),
mode="valid")/W, label="SMDP")
plt.plot(np.convolve(rewards_per_episode_IO, np.ones(W),
mode="valid")/W, label="IO")
plt.title("comparision of SMDP and IO rewards")
plt.xlabel("episodes")
plt.ylabel("rewards")
plt.legend(loc="best")
plt.show()

plt.plot(np.convolve(steps_per_episode_SMDP, np.ones(W),
mode="valid")/W, label="SMDP")
plt.plot(np.convolve(steps_per_episode_IO, np.ones(W),
mode="valid")/W, label="IO")
plt.title("comparision of SMDP and IO steps")
plt.xlabel("episodes")
plt.ylabel("rewards")
plt.legend(loc="best")
plt.show()
```



Alternate Options

```
q_values_SMDP_alt = np.zeros((500,  10))
q_values_IO_alt = np.zeros((500,  10))

# primitive actions
N, S, E, W, P, D = 1, 0, 2, 3, 4, 5

# ALTERNATE options
Opt_North_Alt, Opt_South_Alt, Opt_East_Alt, Opt_West_Alt = 6, 7, 8, 9

# alternate option functions
def North_Alt(r, c, p_loc, dest):
    optact = N
    optdone = False
    if [r, c] == R:
        if p_loc == 0:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 0:
            optact = D
            optdone = True
    elif [r, c] == B:
        if p_loc == 3:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 3:
```

```
            optact = D
            optdone = True
    elif [r, c] == G:
        if p_loc == 1:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 1:
            optact = D
            optdone = True
    elif [r, c] == Y:
        if p_loc == 2:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 2:
            optact = D
            optdone = True
    if r == 0:
        optdone = True
    return [optact, optdone]

def South_Alt(r, c, p_loc, dest):
    optact = S
    optdone = False
    if [r, c] == R:
        if p_loc == 0:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 0:
            optact = D
            optdone = True
    elif [r, c] == B:
        if p_loc == 3:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 3:
            optact = D
            optdone = True
    elif [r, c] == G:
        if p_loc == 1:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 1:
            optact = D
            optdone = True
    elif [r, c] == Y:
        if p_loc == 2:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 2:
            optact = D
```

```
            optdone = True
    if r == 4:
        optdone = True
    return [optact, optdone]

def East_Alt(r, c, p_loc, dest):
    optact = E
    optdone = False
    if [r, c] == R:
        if p_loc == 0:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 0:
            optact = D
            optdone = True
    elif [r, c] == B:
        if p_loc == 3:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 3:
            optact = D
            optdone = True
    elif [r, c] == G:
        if p_loc == 1:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 1:
            optact = D
            optdone = True
    elif [r, c] == Y:
        if p_loc == 2:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 2:
            optact = D
            optdone = True
    if (c == 4 or [r, c] == [0, 1] or [r, c] == [3, 0] or [r, c] ==
[4, 0] or [r, c] == [3, 2] or [r, c] == [4, 2] or [r, c] == [1, 1]):
        optdone = True
    return [optact, optdone]

def West_Alt(r, c, p_loc, dest):
    optact = W
    optdone = False
    if [r, c] == R:
        if p_loc == 0:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 0:
            optact = D
```

```
            optdone = True
    elif [r, c] == B:
        if p_loc == 3:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 3:
            optact = D
            optdone = True
    elif [r, c] == G:
        if p_loc == 1:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 1:
            optact = D
            optdone = True
    elif [r, c] == Y:
        if p_loc == 2:
            optact = P
            optdone = True
        elif p_loc == 4 and dest == 2:
            optact = D
            optdone = True
    if (c == 0 or [r, c] == [0, 2] or [r, c] == [3, 1] or [r, c] ==
[4, 1] or [r, c] == [3, 3] or [r, c] == [4, 3] or [r, c] == [1, 2]):
        optdone = True
    return [optact, optdone]
```

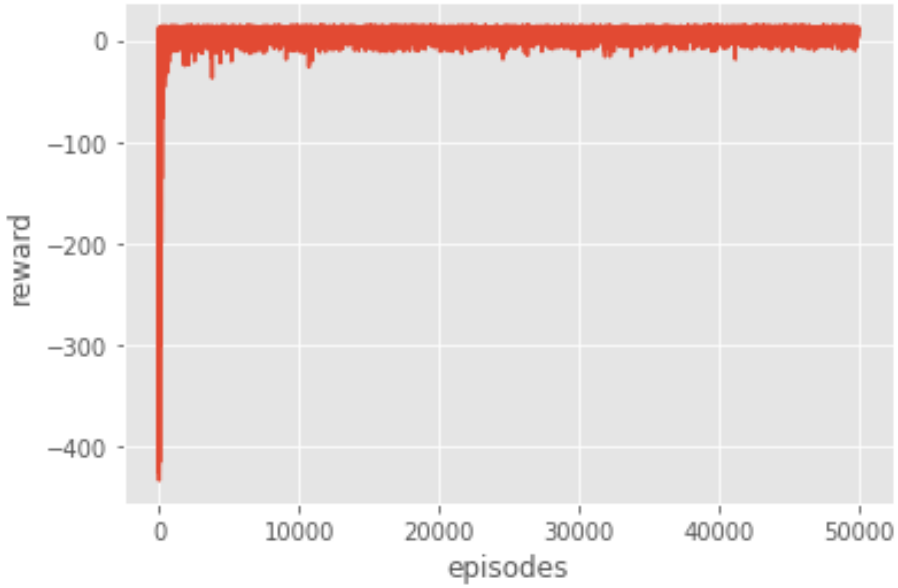
SMDP Q-Learning with Alternate Options

```
gamma, lr = 0.99, 0.75
rewards_per_episode_SMDP_alt, steps_per_episode_SMDP_alt = [], []
action_freq_SMDP_alt = [0]*10

for _ in range(50000):
    state = env.reset()
    done = False
    total_reward = 0
    total_steps = 0

    while not done:
        reward_bar, T = 0, 0
        action = egreedy_policy(q_values_SMDP_alt, state,
epsilon=0.01)
        action_freq_SMDP_alt[action] += 1

        # primitive action
        if action < 6:
            next_state, reward, done, _ = env.step(action)
            total_reward += reward
            q_values_SMDP_alt[state, action] += (lr * (reward + (gamma
* q_values_SMDP_alt[next_state].max()) - q_values_SMDP_alt[state,
```

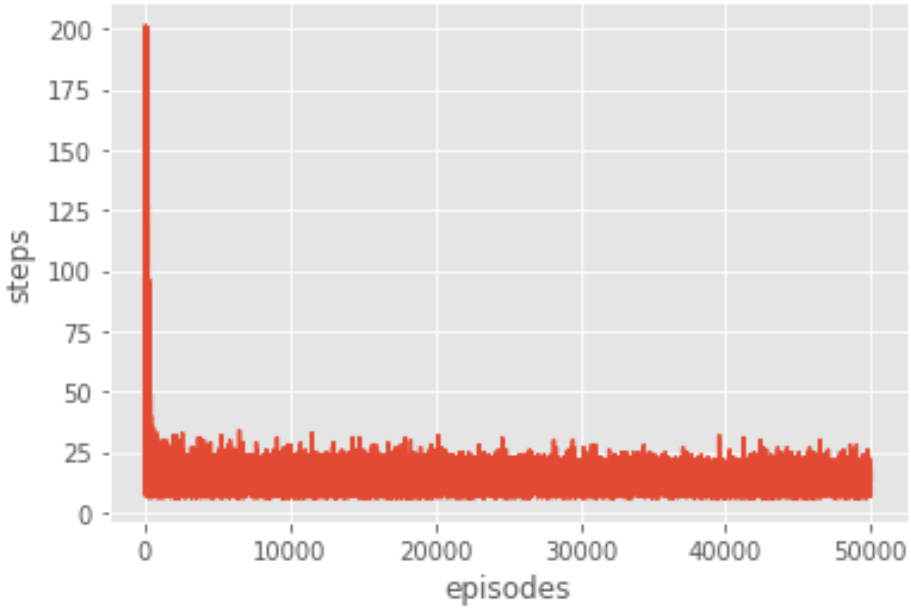
```
action]))
            state = next_state
            total_steps += 1

        # Checking if action chosen is an option
        elif action == Opt_North_Alt: # option to always move North
            optdone = False
            state_t = state
            while not optdone:
                optact, optdone = North_Alt(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                reward_bar  += ((gamma ** T) * reward)
                state = next_state
                total_steps += 1
                T += 1
            total_reward += reward_bar
            q_values_SMDP_alt[state_t, action] += (lr * (reward_bar +
((gamma ** T) * q_values_SMDP_alt[state].max()) -
q_values_SMDP_alt[state_t, action]))

        elif action == Opt_South_Alt: # option to always move South
            optdone = False
            state_t = state
            while not optdone:
                optact, optdone = South_Alt(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                reward_bar  += ((gamma ** T) * reward)
                state = next_state
                total_steps += 1
                T += 1
            total_reward += reward_bar
            q_values_SMDP_alt[state_t, action] += (lr * (reward_bar +
((gamma ** T) * q_values_SMDP_alt[state].max()) -
q_values_SMDP_alt[state_t, action]))

        elif action == Opt_East_Alt: # option to always move East
            optdone = False
            state_t = state
            while not optdone:
                optact, optdone = East_Alt(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                reward_bar  += ((gamma ** T) * reward)
                state = next_state
                total_steps += 1
                T += 1
            total_reward += reward_bar
            q_values_SMDP_alt[state_t, action] += (lr * (reward_bar +
((gamma ** T) * q_values_SMDP_alt[state].max()) -
q_values_SMDP_alt[state_t, action]))
```

```
        elif action == Opt_West_Alt: # option to always move West
            optdone = False
            state_t = state
            while not optdone:
                optact, optdone = West_Alt(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                reward_bar  += ((gamma ** T) * reward)
                state = next_state
                total_steps += 1
                T += 1
            total_reward += reward_bar
            q_values_SMDP_alt[state_t, action] += (lr * (reward_bar +
((gamma ** T) * q_values_SMDP_alt[state].max()) -
q_values_SMDP_alt[state_t, action]))

    rewards_per_episode_SMDP_alt.append(total_reward)
    steps_per_episode_SMDP_alt.append(total_steps)

plt.plot(rewards_per_episode_SMDP_alt)
plt.xlabel("episodes")
plt.ylabel("reward")
plt.title("rewards per episode with SMDP Q-Learning with alternate
options")
plt.show()

plt.plot(steps_per_episode_SMDP_alt)
plt.xlabel("episodes")
plt.ylabel("steps")
plt.title("steps per episode with SMDP Q-Learning with alternate
options")
plt.show()

plt.bar(range(10), action_freq_SMDP_alt, width=0.3)
plt.xlabel("# action")
plt.ylabel("frequency")
plt.title("Frequency of the actions being chosen with SMDP (alternate
options)")
plt.show()

plt.figure(figsize=(15, 100))
plt.pcolor(q_values_SMDP_alt, edgecolors='k', linewidths=0.5,
cmap="viridis")
plt.title("Heat-Map for SMDP Q-Learning with alternate options")
plt.xticks(np.arange(0, 11, 1.0))
plt.colorbar(shrink=0.1)
plt.show()
```

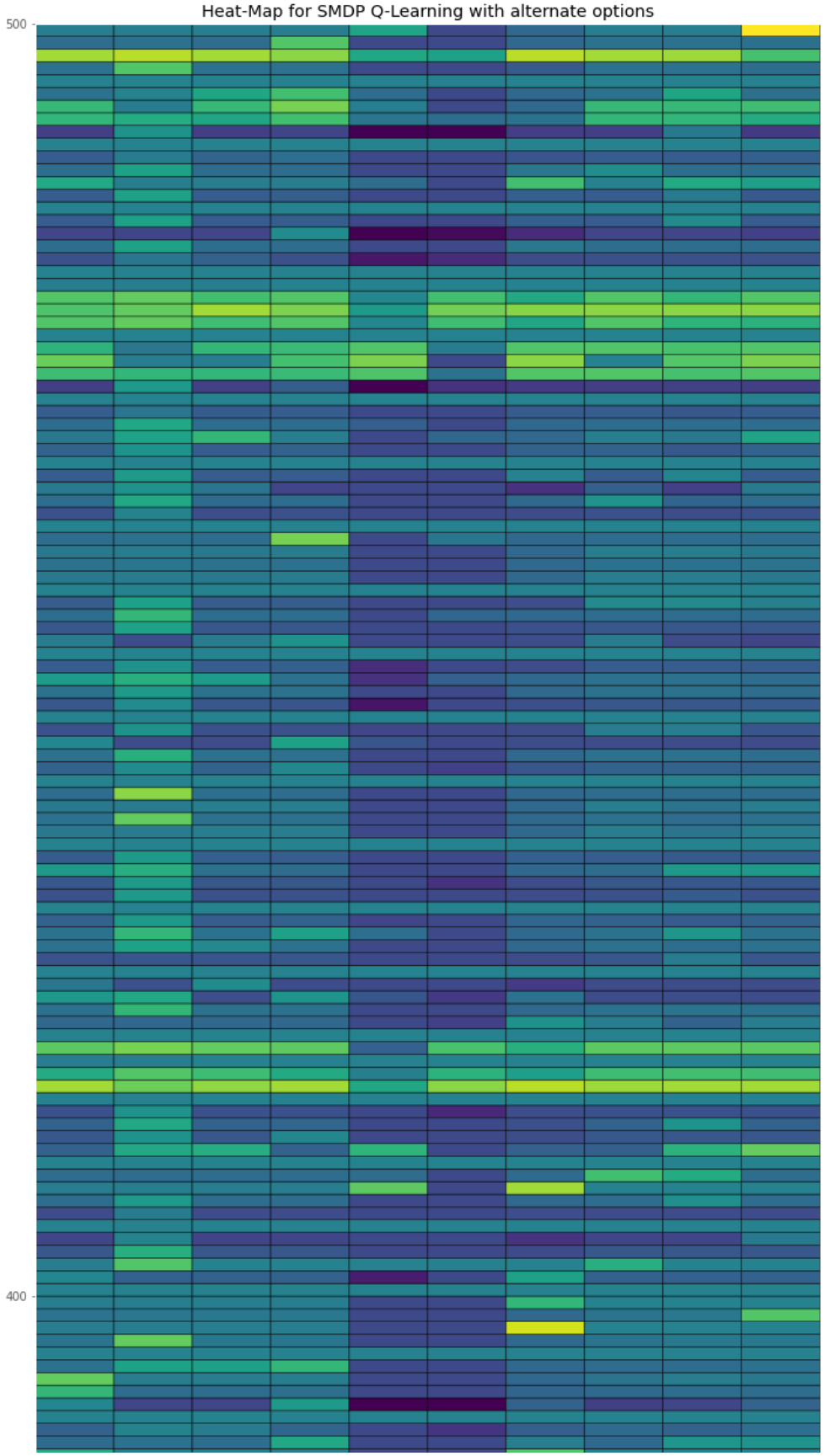
rewards per episode with SMDP Q-Learning with alternate options



steps per episode with SMDP Q-Learning with alternate options




```
C:\Users\Varun Gumma\AppData\Local\Temp\
ipykernel_5512\1317524724.py:20: MatplotlibDeprecationWarning: Auto-
removal of grids by pcolor() and pcolormesh() is deprecated since 3.5
and will be removed two minor releases later; please call grid(False)
first.
  plt.pcolor(q_values_SMDP_alt, edgecolors='k', linewidths=0.5,
cmap="viridis")
```

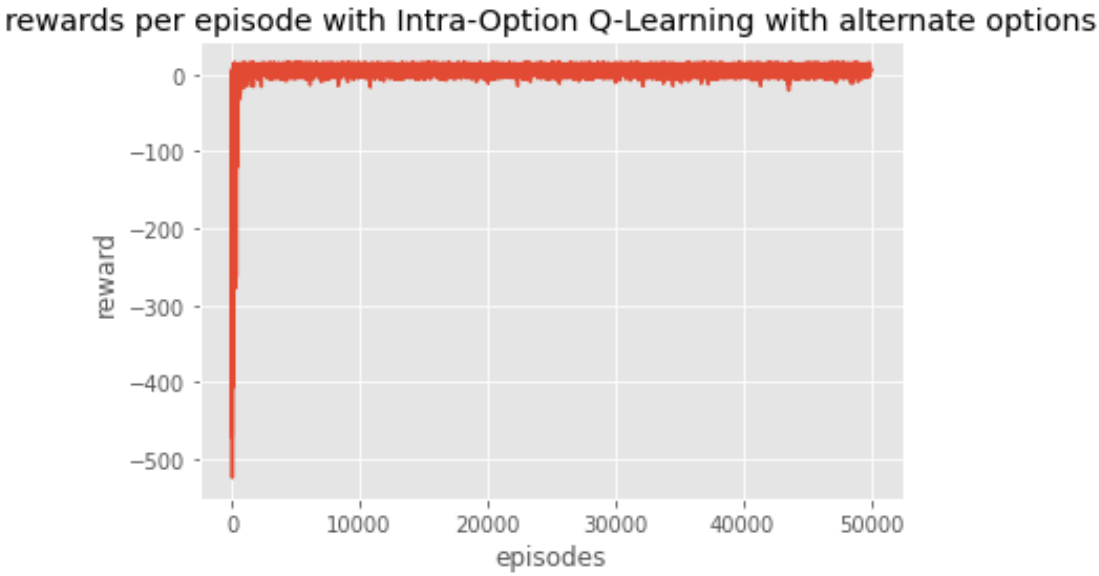

Heat-Map for SMDP Q-Learning with alternate options
500
400

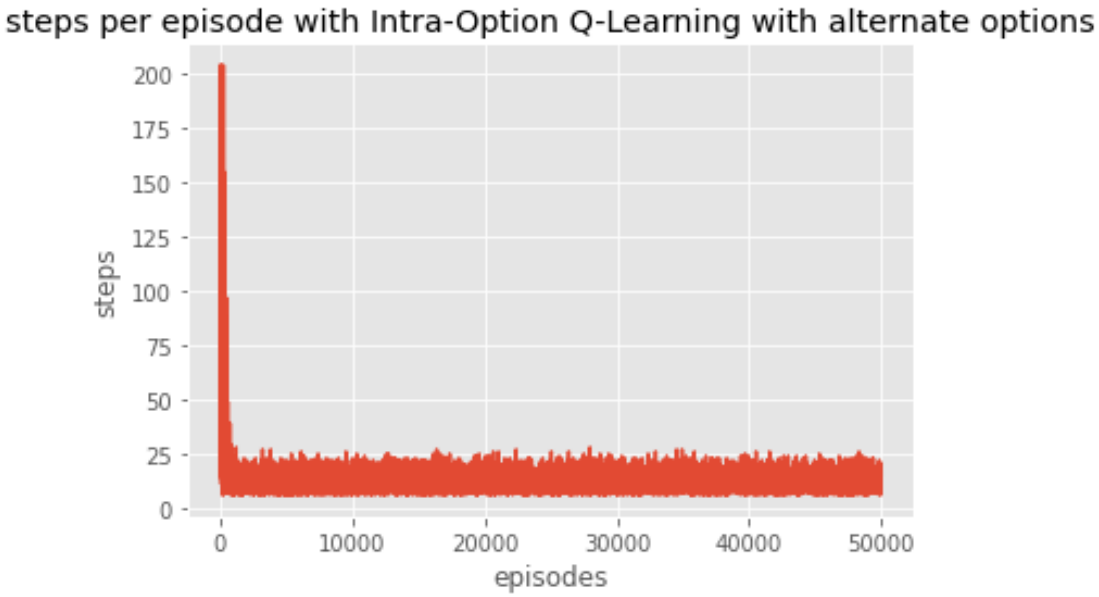
Intra-Option Q-Learning with Alternate Options

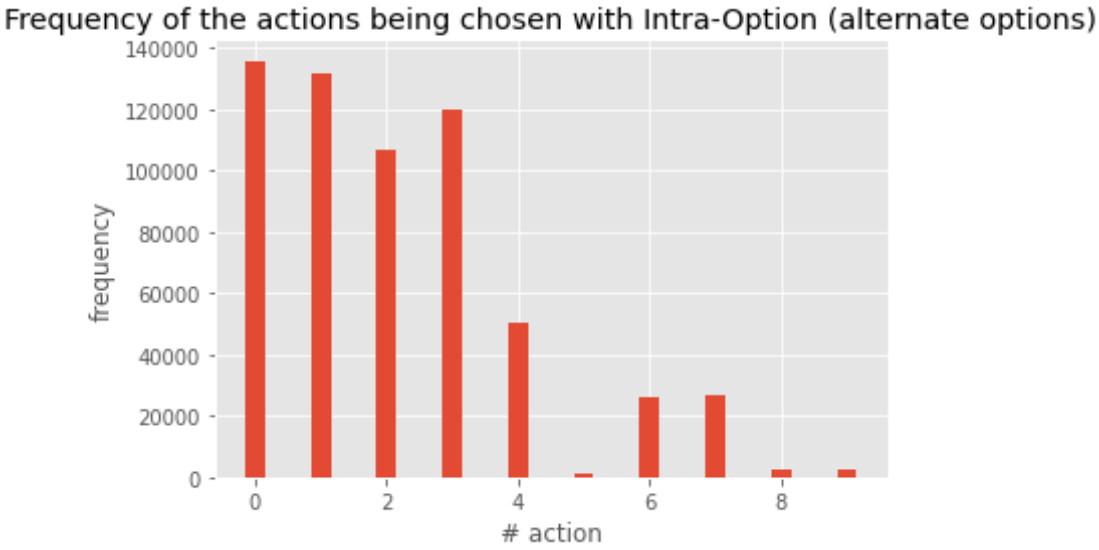
```
gamma, lr = 0.99, 0.75
rewards_per_episode_IO_alt, steps_per_episode_IO_alt = [], []
action_freq_IO_alt = [0]*10

for _ in range(50000):
    state = env.reset()
    done = False
    total_reward = 0
    total_steps = 0

    while not done:
        action = egreedy_policy(q_values_IO_alt, state, epsilon=0.01)
        action_freq_IO_alt[action] += 1

        # Checking if primitive action
        if action < 6:
            next_state, reward, done, _ = env.step(action)
            q_values_IO_alt[state, action] += (lr * (reward + (gamma *
q_values_IO_alt[next_state].max()) - q_values_IO_alt[state, action]))
            state = next_state
            total_reward += reward
            total_steps += 1

        # Checking if action chosen is an option
        elif action == Opt_North_Alt: # option to always move North
            optdone = False
            while not optdone:
                optact, optdone = North_Alt(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                q = q_values_IO_alt[next_state].max() if
is_terminating(next_state) else q_values_IO_alt[next_state, action]
                q_values_IO_alt[state, action] += (lr * (reward +
(gamma * q) - q_values_IO_alt[state, action]))
                state = next_state
                total_reward += reward
                total_steps += 1

        elif action == Opt_South_Alt: # option to always move South
            optdone = False
            while not optdone:
                optact, optdone = South_Alt(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                q = q_values_IO_alt[next_state].max() if
is_terminating(next_state) else q_values_IO_alt[next_state, action]
                q_values_IO_alt[state, action] += (lr * (reward +
(gamma * q) - q_values_IO_alt[state, action]))
                state = next_state
                total_reward += reward
                total_steps += 1
```

```
        elif action == Opt_West_Alt: # option to always move West
            optdone = False
            while not optdone:
                optact, optdone = West_Alt(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                q = q_values_IO_alt[next_state].max() if
is_terminating(next_state) else q_values_IO_alt[next_state, action]
                q_values_IO_alt[state, action] += (lr * (reward +
(gamma * q) - q_values_IO_alt[state, action]))
                state = next_state
                total_reward += reward
                total_steps += 1

        elif action == Opt_East_Alt: # option to always move East
            optdone = False
            while not optdone:
                optact, optdone = East_Alt(*env.decode(state))
                next_state, reward, done, _ = env.step(optact)
                q = q_values_IO_alt[next_state].max() if
is_terminating(next_state) else q_values_IO_alt[next_state, action]
                q_values_IO_alt[state, action] += (lr * (reward +
(gamma * q) - q_values_IO_alt[state, action]))
                state = next_state
                total_reward += reward
                total_steps += 1

    rewards_per_episode_IO_alt.append(total_reward)
    steps_per_episode_IO_alt.append(total_steps)

plt.plot(rewards_per_episode_IO_alt)
plt.xlabel("episodes")
plt.ylabel("reward")
plt.title("rewards per episode with Intra-Option Q-Learning with
alternate options")
plt.show()

plt.plot(steps_per_episode_IO_alt)
plt.xlabel("episodes")
plt.ylabel("steps")
plt.title("steps per episode with Intra-Option Q-Learning with
alternate options")
plt.show()

plt.bar(range(10), action_freq_IO_alt, width=0.3)
plt.xlabel("# action")
plt.ylabel("frequency")
plt.title("Frequency of the actions being chosen with Intra-Option
(alternate options)")
plt.show()
```

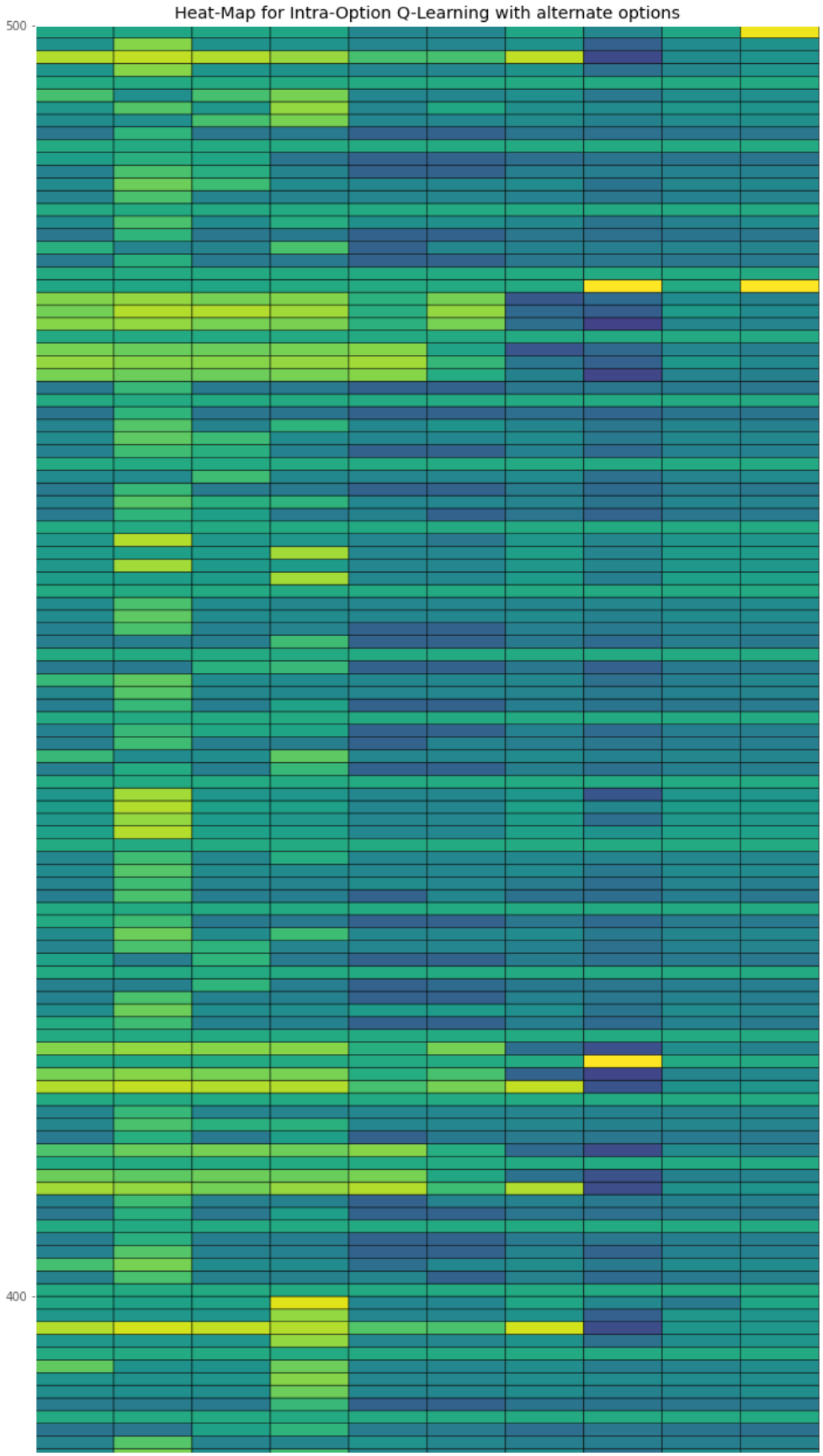
```
plt.figure(figsize=(15, 100))
plt.pcolor(q_values_IO_alt, edgecolors='k', linewidths=0.5,
cmap="viridis")
plt.title("Heat-Map for Intra-Option Q-Learning with alternate
options")
plt.xticks(np.arange(0, 11, 1.0))
plt.colorbar(shrink=0.1)
plt.show()
```



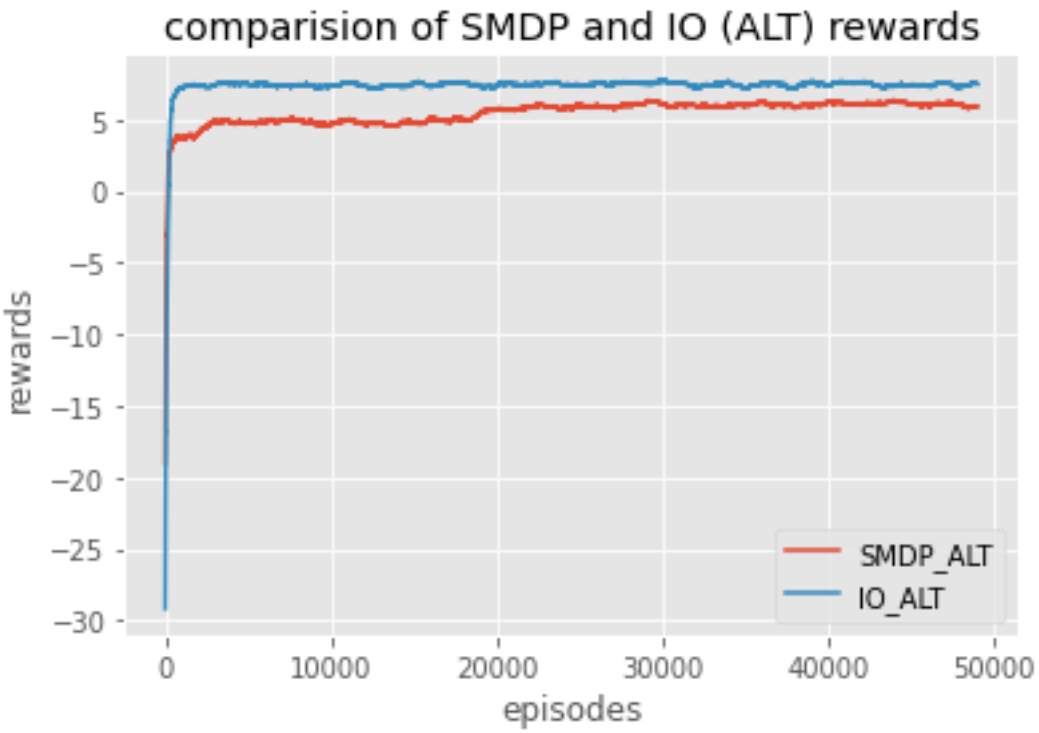


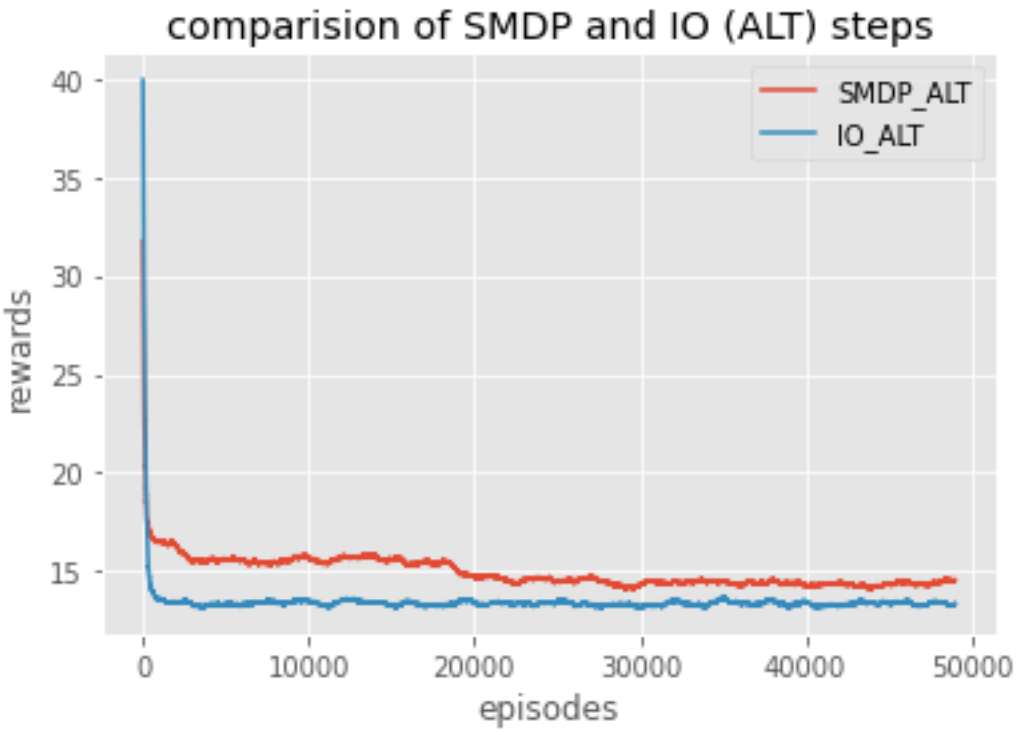



```
C:\Users\Varun Gumma\AppData\Local\Temp\
ipykernel_5512\2324895190.py:20: MatplotlibDeprecationWarning: Auto-
removal of grids by pcolor() and pcolormesh() is deprecated since 3.5
and will be removed two minor releases later; please call grid(False)
first.
  plt.pcolor(q_values_IO_alt, edgecolors='k', linewidths=0.5,
cmap="viridis")
```


Heat-Map for Intra-Option Q-Learning with alternate options
500
400

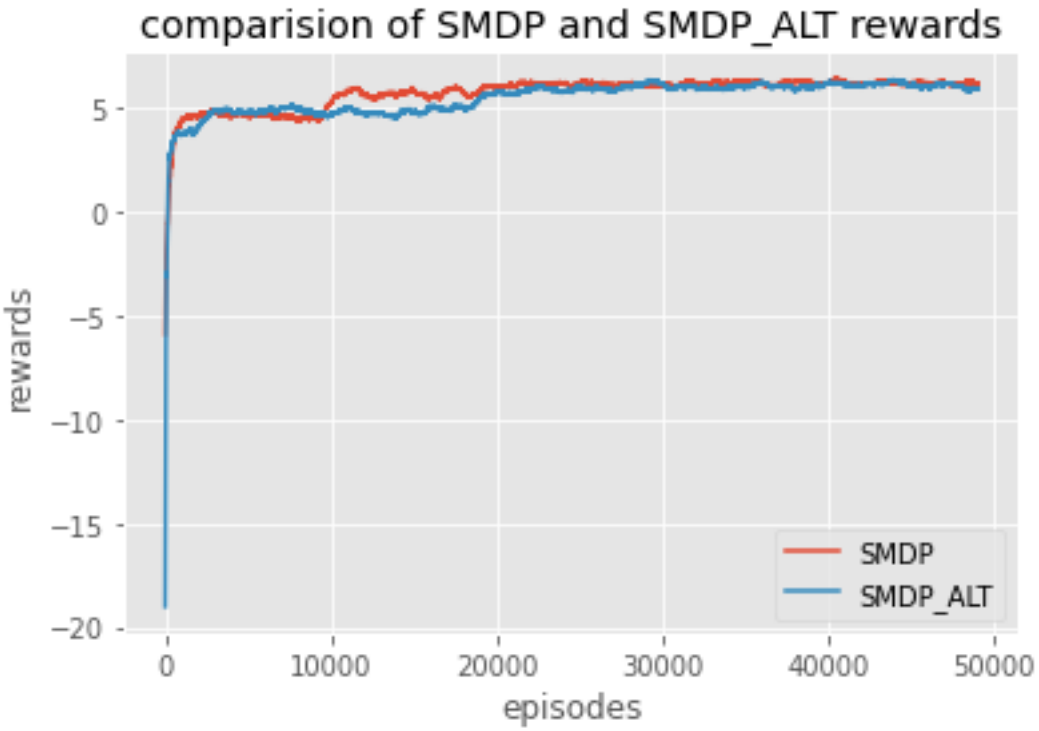
```
W = 1000
plt.plot(np.convolve(rewards_per_episode_SMDP_alt, np.ones(W),
mode="valid")/W, label="SMDP_ALT")
plt.plot(np.convolve(rewards_per_episode_IO_alt, np.ones(W),
mode="valid")/W, label="IO_ALT")
plt.title("comparision of SMDP and IO (ALT) rewards")
plt.xlabel("episodes")
plt.ylabel("rewards")
plt.legend(loc="best")
plt.show()

plt.plot(np.convolve(steps_per_episode_SMDP_alt, np.ones(W),
mode="valid")/W, label="SMDP_ALT")
plt.plot(np.convolve(steps_per_episode_IO_alt, np.ones(W),
mode="valid")/W, label="IO_ALT")
plt.title("comparision of SMDP and IO (ALT) steps")
plt.xlabel("episodes")
plt.ylabel("rewards")
plt.legend(loc="best")
plt.show()
```

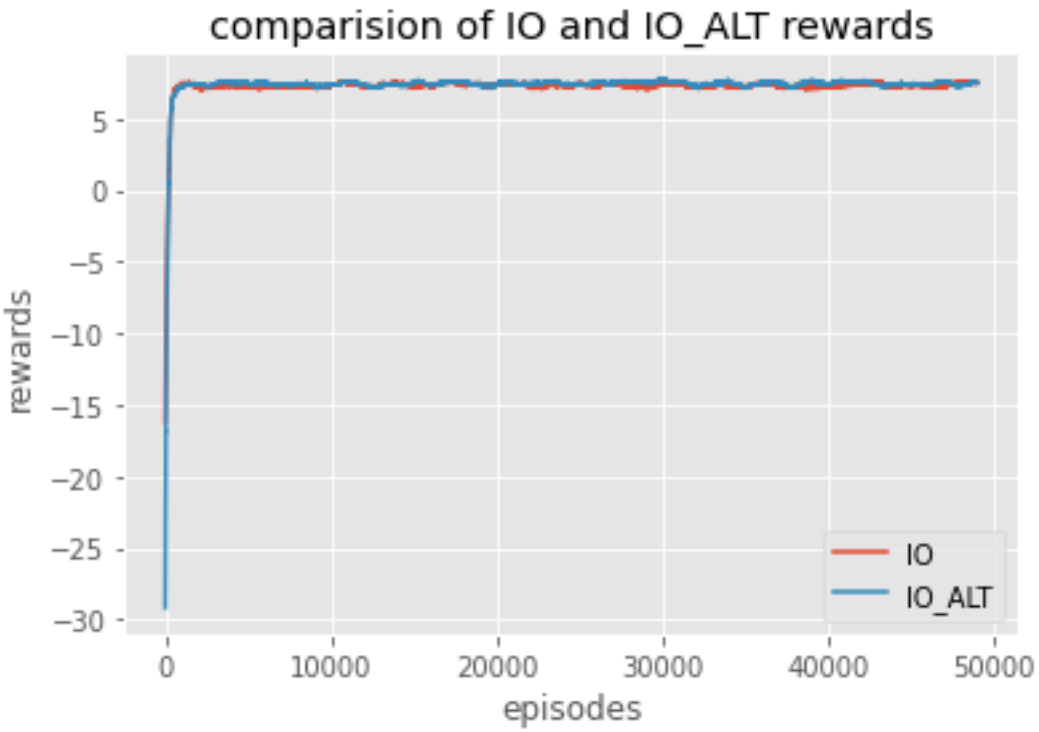


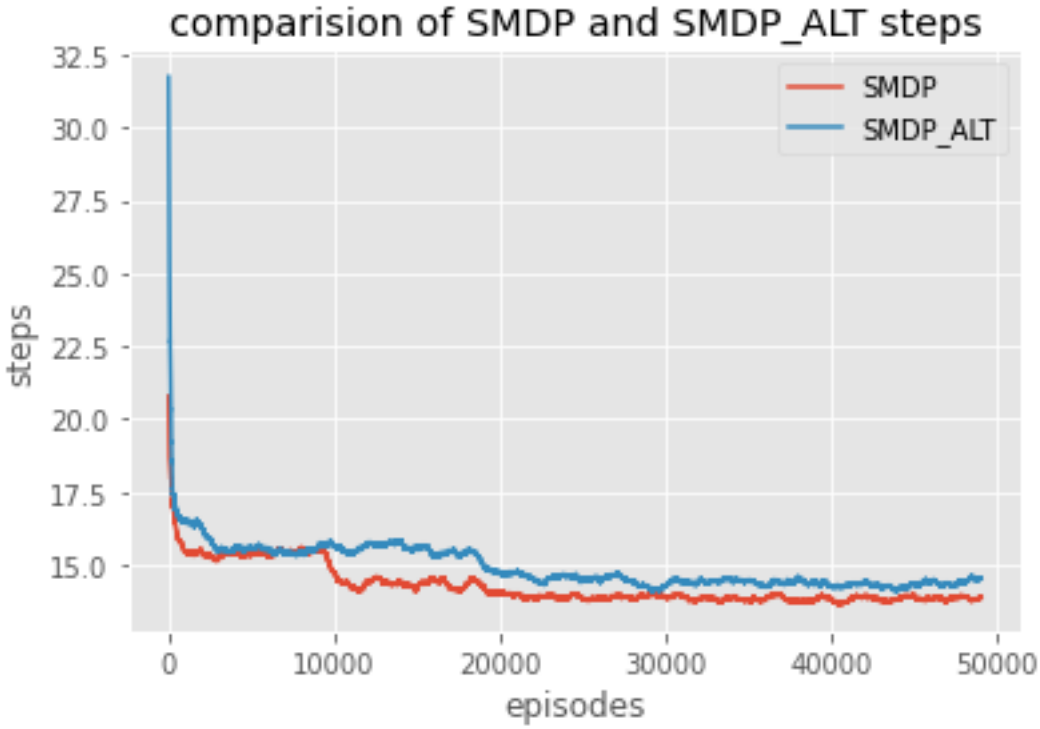


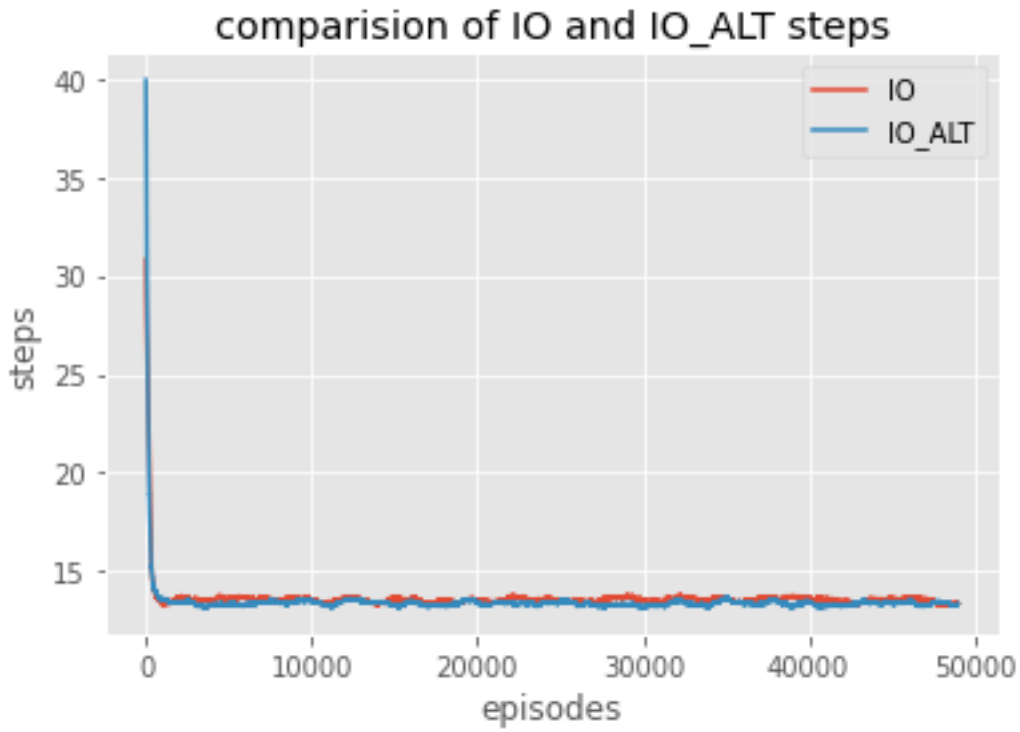

```
W = 1000
plt.plot(np.convolve(rewards_per_episode_SMDP, np.ones(W),
mode="valid")/W, label="SMDP")
plt.plot(np.convolve(rewards_per_episode_SMDP_alt, np.ones(W),
mode="valid")/W, label="SMDP_ALT")
plt.title("comparision of SMDP and SMDP_ALT rewards")
plt.xlabel("episodes")
plt.ylabel("rewards")
plt.legend(loc="best")
plt.show()

plt.plot(np.convolve(rewards_per_episode_IO, np.ones(W),
mode="valid")/W, label="IO")
plt.plot(np.convolve(rewards_per_episode_IO_alt, np.ones(W),
mode="valid")/W, label="IO_ALT")
plt.title("comparision of IO and IO_ALT rewards")
plt.xlabel("episodes")
plt.ylabel("rewards")
plt.legend(loc="best")
plt.show()

plt.plot(np.convolve(steps_per_episode_SMDP, np.ones(W),
mode="valid")/W, label="SMDP")
plt.plot(np.convolve(steps_per_episode_SMDP_alt, np.ones(W),
mode="valid")/W, label="SMDP_ALT")
plt.title("comparision of SMDP and SMDP_ALT steps")
plt.xlabel("episodes")
```

```
plt.ylabel("steps")
plt.legend(loc="best")
plt.show()

plt.plot(np.convolve(steps_per_episode_IO, np.ones(W),
mode="valid")/W, label="IO")
plt.plot(np.convolve(steps_per_episode_IO_alt, np.ones(W),
mode="valid")/W, label="IO_ALT")
plt.title("comparision of IO and IO_ALT steps")
plt.xlabel("episodes")
plt.ylabel("steps")
plt.legend(loc="best")
plt.show()
```




comparision of IO and IO_ALT rewards
rewards
episodes
IO
IO_ALT
5
0
−5
−10
−15
−20
−25
−30
0
10000
20000
30000
40000
50000


comparision of SMDP and SMDP_ALT steps
steps
episodes
SMDP
SMDP_ALT
32.5
30.0
27.5
25.0
22.5
20.0
17.5
15.0
0
10000
20000
30000
40000
50000


comparision of IO and IO_ALT steps
IO
IO_ALT
steps
episodes
40
35
30
25
20
15
0
10000
20000
30000
40000
50000