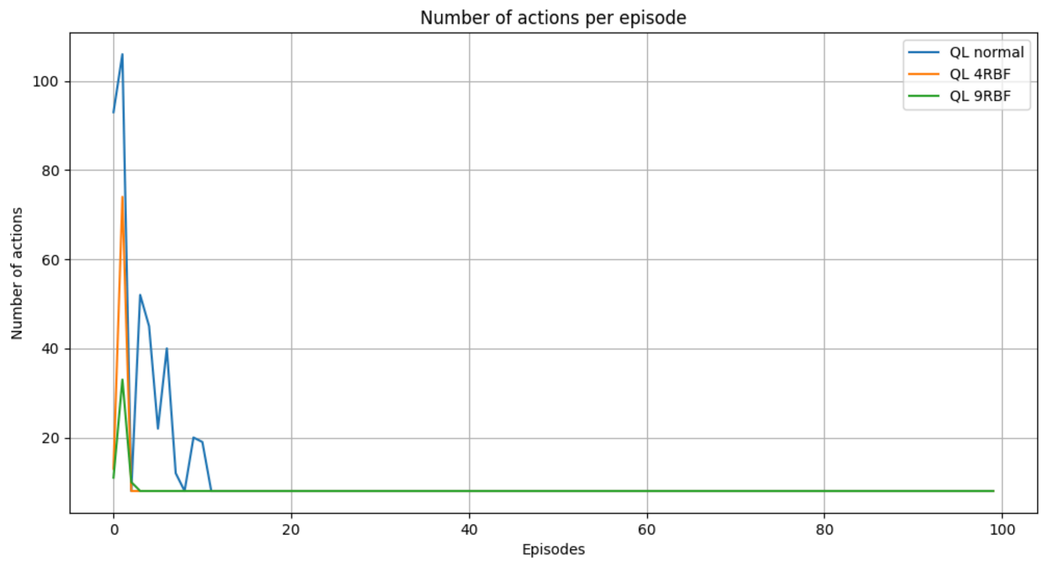
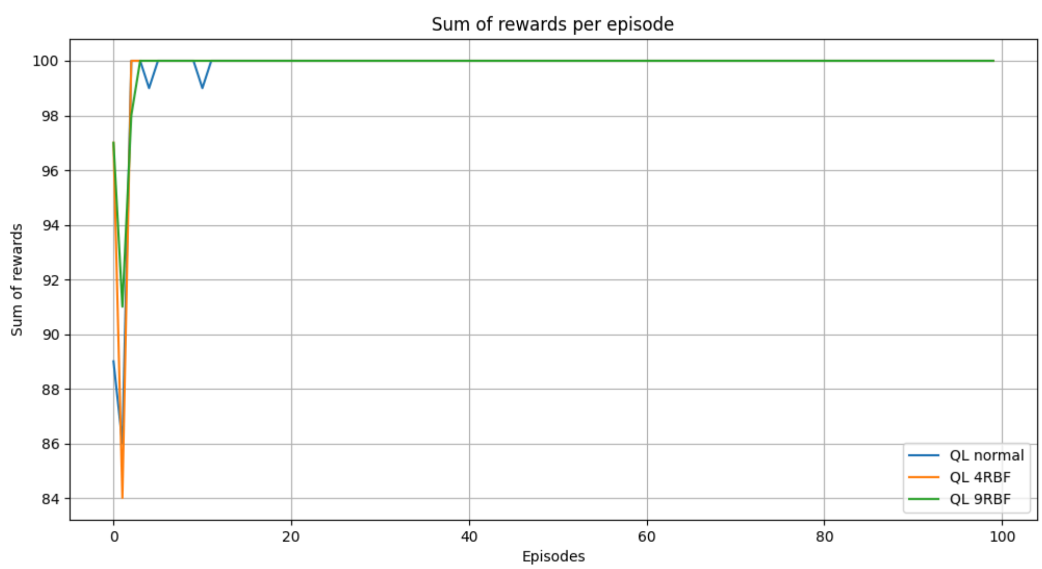
1. Plot the number of actions the robot takes in each episode for QL, RBF-QL w/ 4BF, and RBF-QL w/ 9BF



2. Plot the reward of the robot in each episode for QL, RBF-QL w/ 4BF, and RBF-QL w/ 9BF



Code:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import random
import math

# functions for normal QL
def
choose_action_QL_normal(s,Q_table,epsilon,p_array,iteration,episode)
:
    # select the action with highest estimated action value
    # if several actions with same Q value: randomize
    # get maximum value
    max_action_value = max(Q_table[s].values())
    # get all keys with maximum value
    max_action_keys = [key for key,value in Q_table[s].items() if
value == max_action_value]
    # decaying epsilon. Higher epsilon at start of training for more
exploration. In later episodes uses exploitation.
    if episode == 0 or episode == 1:
        epsilon = 0.5
    elif episode < 10:
        epsilon = 0.01
    else:
        epsilon = 0.0
    p = p_array[iteration]
    if p >= epsilon:
        if len(max_action_keys) > 1:
            action = random.choice(max_action_keys)
        else:
            action = max_action_keys[0]
    else:
    # randomize
        action = random.choice(list(Q_table[s]))
    return action

def take_action_QL_normal(action, s):
# set new state s'
    if action == 'up':
        sprime = (s[0]-1, s[1])
    if action == 'down':
        sprime = (s[0]+1, s[1])
    if action == 'left':
```

```
        sprime = (s[0], s[1]-1)
    if action == 'right':
        sprime = (s[0], s[1]+1)
    # set reward r
    # • Action that makes the robot tend to go out of the grid will
get a reward of -1 (when the robot is in the border cells)
    # • Action that makes the robot reach the goal will get a reward
of 100
    # • All other actions will get a reward of 0
    if sprime == (4,4):
        r = 100
    elif sprime[0] == -1 or sprime[1] == -1 or sprime[0] == 5 or
sprime[1] == 5:
        r = -1
    else:
        r = 0
    # if action would be out of the border, robot stays in current
cell
    if r == -1:
        sprime = s
    return r, sprime

def QL_normal(gamma, alpha, epsilon, p_array, s, Q_table, episode):
    reward_sum_episode_QL_normal = 0
    action_sum_episode_QL_normal = 0
    iteration = 0
    iteration_terminate = 0
    """ Repeat (for each step of episode): """
    # while loop until goal is reached
    while s != (4,4) and iteration_terminate < 10000:
        """ Choose a from s using policy derived from Q (e.g.
epsilon-greedy) """
        action =
choose_action_QL_normal(s,Q_table,epsilon,p_array,iteration,
episode)
        """ Take action a, observe r,s' """
        r, sprime = take_action_QL_normal(action, s)
        """ Q[s,action] += alpha * (reward + (gamma *
predicted_value) - Q[s,action]) """
        predicted_value = max(Q_table[sprime].values(), default=0)
```

```
        Q_table[s][action] += alpha * (r + (gamma * predicted_value)
- Q_table[s][action])
        action_sum_episode_QL_normal += 1
        reward_sum_episode_QL_normal += r
        if iteration < 199:
            iteration += 1
        else:
            iteration = 0
        iteration_terminate += 1
        """ s = s' """
        s = sprime
        """ Until s is terminal """
    return action_sum_episode_QL_normal,
reward_sum_episode_QL_normal

def choose_action_4RBF(s,theta,epsilon,p_array,c,mu,iteration,
episode):
# select the action with highest estimated action value
# if several actions with same value: randomize
# calculate phi for all actions
    phi_all_actions = []
    # for all 4 actions
    for i in range(4):
        phi = np.zeros(16)
        for l in range(4):
            phi[i*4+l] = math.exp( - (np.linalg.norm( s - c[l,:])**2
) /(2*(mu[l]**2)) )
        phi_all_actions.append(phi)
    # calculate phi_transp*theta for each action
    phi_t_mult_theta_list = []
    for phi in phi_all_actions:
        phi_t_mult_theta_list.append(phi@theta)

    max_action_keys = [jj for jj, j in enumerate(
phi_t_mult_theta_list ) if j == max( phi_t_mult_theta_list )]
    # decaying epsilon. Higher epsilon at start of training for more
exploration. In later episodes uses exploitation.
    if episode == 0 or episode == 1:
        epsilon = 0.5
    elif episode < 10:
```

```
        epsilon = 0.01
    else:
        epsilon = 0.0
    p = p_array[iteration]
    if p >= epsilon:
    # if more than one maximum value action found
        if len(max_action_keys) > 1:
        # randomize
            action = random.choice(max_action_keys)
        else:
            action = max_action_keys[0]
    else:
    # randomize
        action = random.randint(0, 3)
    return action


def take_action_4RBF(action, s):
# set new state s'
    if action == 0:
        sprime = (s[0]-1, s[1])
    if action == 1:
        sprime = (s[0]+1, s[1])
    if action == 2:
        sprime = (s[0], s[1]-1)
    if action == 3:
        sprime = (s[0], s[1]+1)
    # set reward r
    # • Action that makes the robot tend to go out of the grid will
get a reward of -1 (when the robot is in the border cells)
    # • Action that makes the robot reach the goal will get a reward
of 100
    # • All other actions will get a reward of 0
    if sprime == (4,4):
        r = 100
    elif sprime[0] == -1 or sprime[1] == -1 or sprime[0] == 5 or
sprime[1] == 5:
        r = -1
    else:
        r = 0
```

```
    # if action would be out of the border, robot stays in current
cell
    if r == -1:
        sprime = s
    return r, sprime

def QL_4RBF(gamma, alpha, epsilon, p_array, s, c, mu, theta,
episode):
    reward_sum_episode_4RBF = 0
    action_sum_episode_4RBF = 0
    iteration = 0
    iteration_terminate = 0
    """ Repeat (for each step of episode): """
    # while loop until goal is reached
    while s != (4,4) and iteration_terminate < 10000:
        """ Choose a from A using greedy policy with probability p
"""
        action =
choose_action_4RBF(s,theta,epsilon,p_array,c,mu,iteration, episode)
        """ Take action a, observe r,s' """
        r, sprime = take_action_4RBF(action, s)
        """ Estimate phi_s """
        phi_s = np.zeros(16)
        for i in range(4):
            if action == i:
                for l in range(4):
                    phi_s[i*4+l] = math.exp( - (np.linalg.norm( s -
c[l,:])**2 ) /(2*(mu[l]**2)) )
        """ Update """
        # calculate predicted value
        # calculate phi_sprime for all actions
        phi_sprime_all_actions = []
        # for all 4 actions
        for i in range(4):
            phi_sprime = np.zeros(16)
            for l in range(4):
                phi_sprime[i*4+l] = math.exp( - (np.linalg.norm(
sprime - c[l,:])**2 ) /(2*(mu[l]**2)) )
            phi_sprime_all_actions.append(phi_sprime)
        # calculate phi_sprime_transp*theta for each action
```

```
        phi_sprime_t_mult_theta_list = []
        for phi_sprime in phi_sprime_all_actions:
            phi_sprime_t_mult_theta_list.append(phi_sprime@theta)
        predicted_value = max(phi_sprime_t_mult_theta_list)
        # theta update
        theta += alpha * (r + (gamma * predicted_value) -
phi_s@theta) * phi_s
        action_sum_episode_4RBF += 1
        reward_sum_episode_4RBF += r
        if iteration < 199:
            iteration += 1
        else:
            iteration = 0
        iteration_terminate += 1
        """ s = s' """
        s = sprime
        """ Until s is terminal """
    return action_sum_episode_4RBF, reward_sum_episode_4RBF

def choose_action_9RBF(s,theta,epsilon,p_array,c,mu,iteration,
episode):
# select the action with highest estimated action value
# if several actions with same value: randomize
# calculate phi for all actions
    phi_all_actions = []
    # for all 4 actions
    for i in range(4):
        phi = np.zeros(36)
        for l in range(9):
            phi[i*9+l] = math.exp( - (np.linalg.norm( s - c[l,:])**2
) /(2*(mu[l]**2)) )
        phi_all_actions.append(phi)
    # calculate phi_transp*theta for each action
    phi_t_mult_theta_list = []
    for phi in phi_all_actions:
        phi_t_mult_theta_list.append(phi@theta)

    max_action_keys = [jj for jj, j in enumerate(
phi_t_mult_theta_list ) if j == max( phi_t_mult_theta_list )]
```

```
    # decaying epsilon. Higher epsilon at start of training for more
exploration. In later episodes uses exploitation.
    if episode == 0 or episode == 1:
        epsilon = 0.5
    elif episode < 10:
        epsilon = 0.01
    else:
        epsilon = 0.0
    p = p_array[iteration]
    if p >= epsilon:
    # if more than one maximum value action found
        if len(max_action_keys) > 1:
        # randomize
            action = random.choice(max_action_keys)
        else:
            action = max_action_keys[0]
    else:
    # randomize
        action = random.randint(0, 3)
    return action


def take_action_9RBF(action, s):
# set new state s'
    if action == 0:
        sprime = (s[0]-1, s[1])
    if action == 1:
        sprime = (s[0]+1, s[1])
    if action == 2:
        sprime = (s[0], s[1]-1)
    if action == 3:
        sprime = (s[0], s[1]+1)
    # set reward r
    # • Action that makes the robot tend to go out of the grid will
get a reward of -1 (when the robot is in the border cells)
    # • Action that makes the robot reach the goal will get a reward
of 100
    # • All other actions will get a reward of 0
    if sprime == (4,4):
        r = 100
```

```
    elif sprime[0] == -1 or sprime[1] == -1 or sprime[0] == 5 or
sprime[1] == 5:
        r = -1
    else:
        r = 0
    # if action would be out of the border, robot stays in current
cell
    if r == -1:
        sprime = s
    return r, sprime

def QL_9RBF(gamma, alpha, epsilon, p_array, s, c, mu, theta,
episode):
    reward_sum_episode_9RBF = 0
    action_sum_episode_9RBF = 0
    iteration = 0
    iteration_terminate = 0
    """ Repeat (for each step of episode): """
    # while loop until goal is reached
    while s != (4,4) and iteration_terminate < 10000:
        """ Choose a from A using greedy policy with probability p
"""
        action =
choose_action_9RBF(s,theta,epsilon,p_array,c,mu,iteration, episode)
        """ Take action a, observe r,s' """
        r, sprime = take_action_9RBF(action, s)
        """ Estimate phi_s """
        phi_s = np.zeros(36)
        for i in range(4):
            if action == i:
                for l in range(9):
                    phi_s[i*9+l] = math.exp( - (np.linalg.norm( s -
c[l,:])**2 ) /(2*(mu[l]**2)) )
        """ Update """
        # calculate predicted value
        # calculate phi_sprime for all actions
        phi_sprime_all_actions = []
        # for all 4 actions
        for i in range(4):
            phi_sprime = np.zeros(36)
```

```
            for l in range(9):
                phi_sprime[i*9+l] = math.exp( - (np.linalg.norm(
sprime - c[l,:])**2 ) /(2*(mu[l]**2)) )
            phi_sprime_all_actions.append(phi_sprime)
        # calculate phi_sprime_transp*theta for each action
        phi_sprime_t_mult_theta_list = []
        for phi_sprime in phi_sprime_all_actions:
            phi_sprime_t_mult_theta_list.append(phi_sprime@theta)
        predicted_value = max(phi_sprime_t_mult_theta_list)
        # theta update
        theta += alpha * (r + (gamma * predicted_value) -
phi_s@theta) * phi_s
        action_sum_episode_9RBF += 1
        reward_sum_episode_9RBF += r
        if iteration < 199:
            iteration += 1
        else:
            iteration = 0
        iteration_terminate += 1
        """ s = s' """
        s = sprime
        """ Until s is terminal """
    return action_sum_episode_9RBF, reward_sum_episode_9RBF

#plot the number of actions the robot takes in each episode for each
method
def plot_actions(actions_QL_normal, actions_4RBF, actions_9RBF):
    plt.figure(figsize=(12, 6))
    plt.plot(actions_QL_normal, label='QL normal')
    plt.plot(actions_4RBF, label='QL 4RBF')
    plt.plot(actions_9RBF, label='QL 9RBF')
    plt.xlabel('Episodes')
    plt.ylabel('Number of actions')
    plt.title('Number of actions per episode')
    plt.legend()
    plt.grid(True)
    plt.show()

#plot the sum of rewards the robot gets in each episode for each
method
```

```python
def plot_rewards(rewards_QL_normal, rewards_4RBF, rewards_9RBF):
    plt.figure(figsize=(12, 6))
    plt.plot(rewards_QL_normal, label='QL normal')
    plt.plot(rewards_4RBF, label='QL 4RBF')
    plt.plot(rewards_9RBF, label='QL 9RBF')
    plt.xlabel('Episodes')
    plt.ylabel('Sum of rewards')
    plt.title('Sum of rewards per episode')
    plt.legend()
    plt.grid(True)
    plt.show()

# main function
def main():
    # set parameters
    gamma = 0.9
    alpha = 0.89
    epsilon = 1.425

    gamma_9 = 0.9
    alpha_9 = 0.65
    epsilon_9 = 1.2
    # initialize Q-table
    Q_table = {}
    for i in range(5):
        for j in range(5):
            Q_table[(i,j)] = {'up':0, 'down':0, 'left':0, 'right':0}
    # initialize state
    s = (0,0)
    # initialize parameters for 4RBF
    c = np.array([[2,2],[2,4],[4,2],[4,4]])
    mu = np.array([1,1,1,1])
    theta = np.zeros(16)
    #initalize parameters for 9RBF
    c_9 =
np.array([[1,1],[1,5],[2,2],[2,4],[3,3],[4,2],[4,4],[5,1],[5,5]])
    mu_9 = np.array([1,1,1,1,1,1,1,1,1])
    theta_9 = np.zeros(36)

    # initialize p_array for epsilon
```

```
    p_array = np.random.rand(300)
    # initialize lists for plotting
    actions_QL_normal = []
    rewards_QL_normal = []
    actions_4RBF = []
    rewards_4RBF = []
    actions_9RBF = []
    rewards_9RBF = []
    # run 200 episodes
    for i in range(100):
        # QL normal
        action_sum_episode_QL_normal, reward_sum_episode_QL_normal =
QL_normal(gamma, alpha, epsilon, p_array, s, Q_table, i)
        actions_QL_normal.append(action_sum_episode_QL_normal)
        rewards_QL_normal.append(reward_sum_episode_QL_normal)
        # QL 4RBF
        action_sum_episode_4RBF, reward_sum_episode_4RBF =
QL_4RBF(gamma, alpha, epsilon, p_array, s, c, mu, theta, i)
        actions_4RBF.append(action_sum_episode_4RBF)
        rewards_4RBF.append(reward_sum_episode_4RBF)
        #QL 9RBF
        action_sum_episode_9RBF, reward_sum_episode_9RBF =
QL_9RBF(gamma_9, alpha_9, epsilon_9, p_array, s, c_9, mu_9, theta_9,
i)
        actions_9RBF.append(action_sum_episode_9RBF)
        rewards_9RBF.append(reward_sum_episode_9RBF)

    # plot results
    plot_actions(actions_QL_normal, actions_4RBF, actions_9RBF)
    plot_rewards(rewards_QL_normal, rewards_4RBF, rewards_9RBF)
if __name__ == '__main__':
    main()
```