2. Reinforcement Learning

There are 3 files:

1. `maze.py` : defines the `MazeEnv` class, the simulation environment which the Q-learning agent will interact in.
2. `qlearning.py` : defines the `qlearn` function which you will implement, along with several helper functions. Follow the instructions in the file.
3. `plotting_utils.py` : defines several plotting and visualization utilities. In particular, you will use `plot_steps_vs_iters` , `plot_several_steps_vs_iters` , `plot_policy_from_q`

In [146]:
```
from qlearning import qlearn
from maze import MazeEnv, ProbabilisticMazeEnv
from plotting_utils import plot_steps_vs_iters, plot_several_steps_vs_iters, p
lot_policy_from_q
```

1. Basic Q Learning experiments

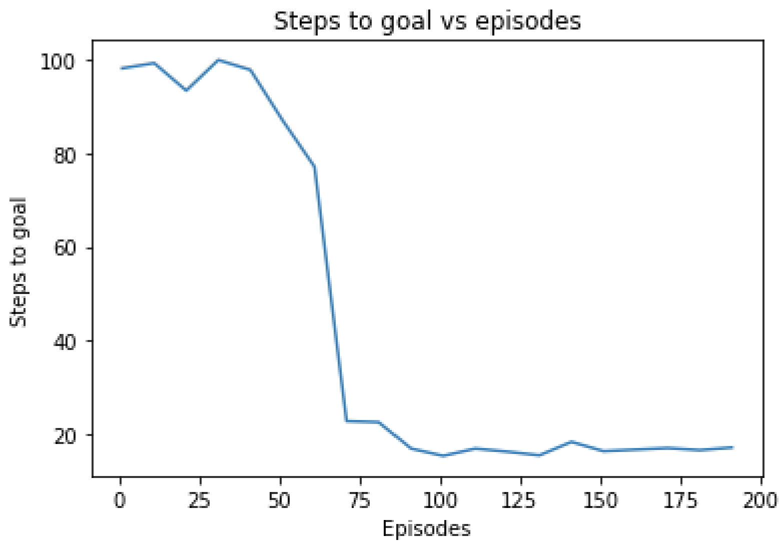
(a) Run your algorithm several times on the given environment. Use the following hyperparameters:

1. Number of episodes = 200
2. Alpha (α) learning rate = 1.0
3. Maximum number of steps per episode = 100. An episode ends when the agent reaches a goal state, or uses the maximum number of steps per episode
4. Gamma (γ) discount factor = 0.9
5. Epsilon (ϵ) for ϵ-greedy = 0.1 (10% of the time). Note that we should "break-ties" when the Q-values are zero for all the actions (happens initially) by essentially choosing uniformly from the action. So now you have two conditions to act randomly: for epsilon amount of the time, or if the Q values are all zero.

In [163]:
```
# TODO: Fill this in
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# TODO: Instantiate the MazeEnv environment with default arguments
env = MazeEnv()

# TODO: Run Q-learning:
q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, max_step
s, use_softmax_policy)
```

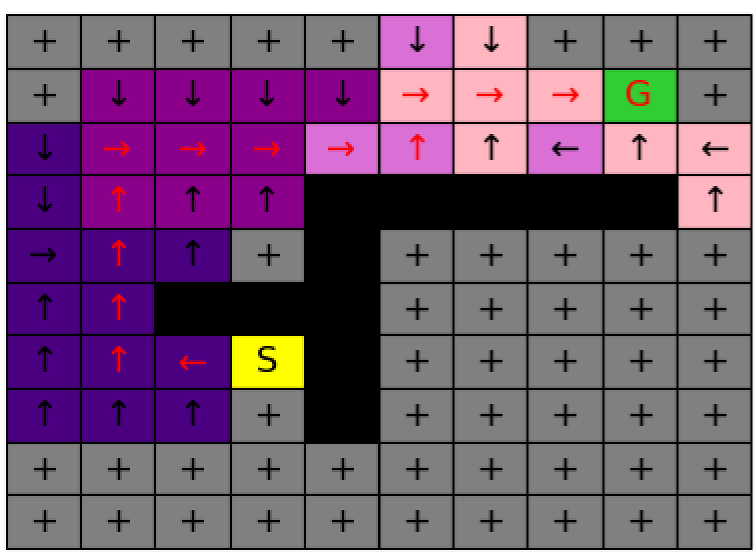
Plot the steps to goal vs training iterations (episodes):

In [164]:
```
# TODO: Plot the steps vs iterations
# plot_steps_vs_iters(...)

plot_steps_vs_iters(steps_vs_iters)
```



Visualize the learned greedy policy from the Q values:

In [139]:
```
# TODO: plot the policy from the Q value
# plot_policy_from_q(...)

plot_policy_from_q(q_hat, env)
```




```
<Figure size 720x720 with 0 Axes>
```

Comments

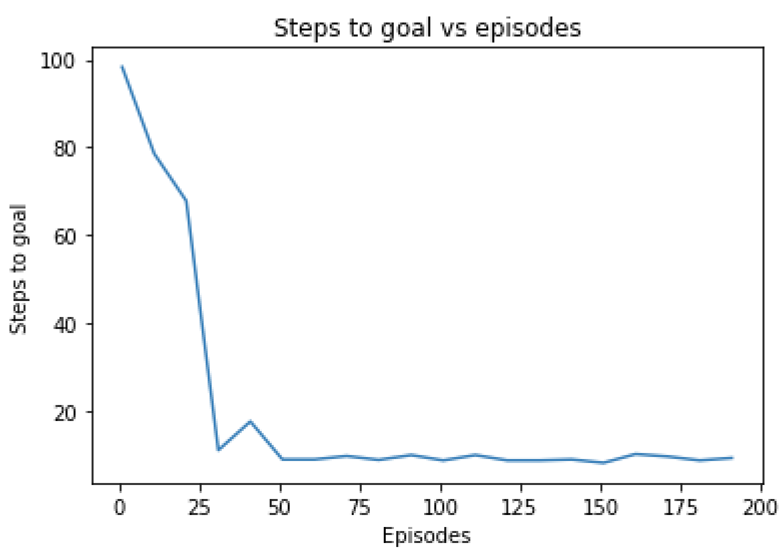
After approximately 75 episodes, the model learns the optimal path to the goal [1,8]. If you see the heat map of the path taken, you can see that it chose the optimal path (taking 14 steps).

(b) Run your algorithm by passing in a list of 2 goal locations: (1,8) and (5,6). Note: we are using 0-indexing, where (0,0) is top left corner. Report on the results.

In [134]:
```
# TODO: Fill this in (same as before)
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# TODO: Set the goal
goal_locs = [[1,8], [5,6]]
env = MazeEnv(goals = goal_locs)

# TODO: Run Q-learning:
q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, max_step
s, use_softmax_policy)
```

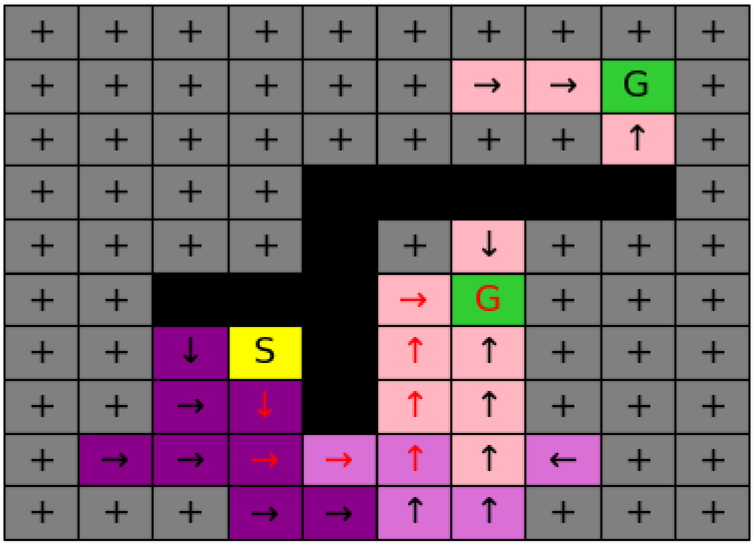
Plot the steps to goal vs training iterations (episodes):

In [135]:
```
# TODO: Plot the steps vs iterations
# plot_steps_vs_iters(...)

plot_steps_vs_iters(steps_vs_iters)
```



Plot the steps to goal vs training iterations (episodes):

In [136]:
```
# TODO: plot the policy from the Q values
# plot_policy_from_q(...)

plot_policy_from_q(q_hat, env)
```




```
<Figure size 720x720 with 0 Axes>
```

Comments

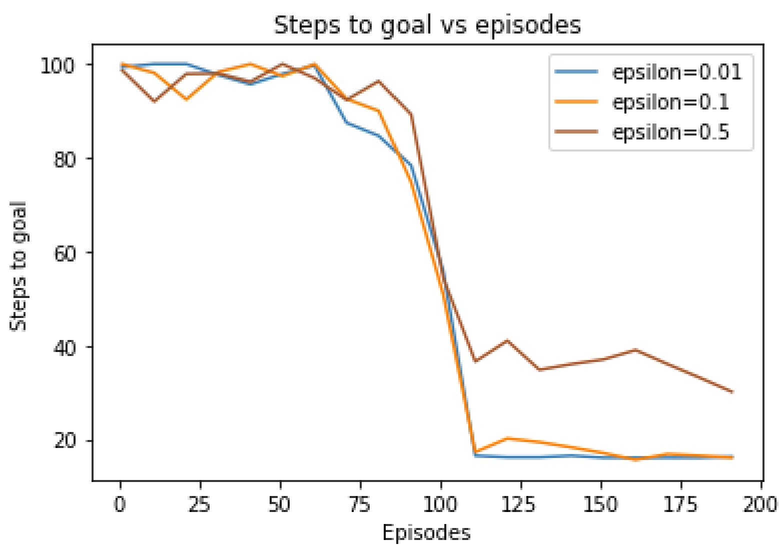
For this part we have two goal locations. One is a lot closer to the start than the other one. Notice that the model took about ~50 episodes before learning the optimal path to the goal. The heatmap shows the optimal route taken to get to the second goal. The model learnt faster than previously because the path is shorter. Heatmap also shows that the model goes to the second goal rather than the first goal (a lot shorter).

2. Experiment with the exploration strategy, in the original environment

(a) Try different ϵ values in ϵ-greedy exploration: We asked you to use a rate of ϵ=10%, but try also 50% and 1%. Graph the results (for 3 epsilon values) and discuss the costs and benefits of higher and lower exploration rates.

In [85]:
```
# TODO: Fill this in (same as before)
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# TODO: set the epsilon lists in increasing order:
epsilon_list = [0.01, 0.1, 0.5]

env = MazeEnv()

steps_vs_iters_list = []
for epsilon in epsilon_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, max_
steps, use_softmax_policy)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [86]:
```
# TODO: Plot the results
label_list = ["epsilon={}".format(eps) for eps in epsilon_list]
# plot_several_steps_vs_iters(...)

plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```



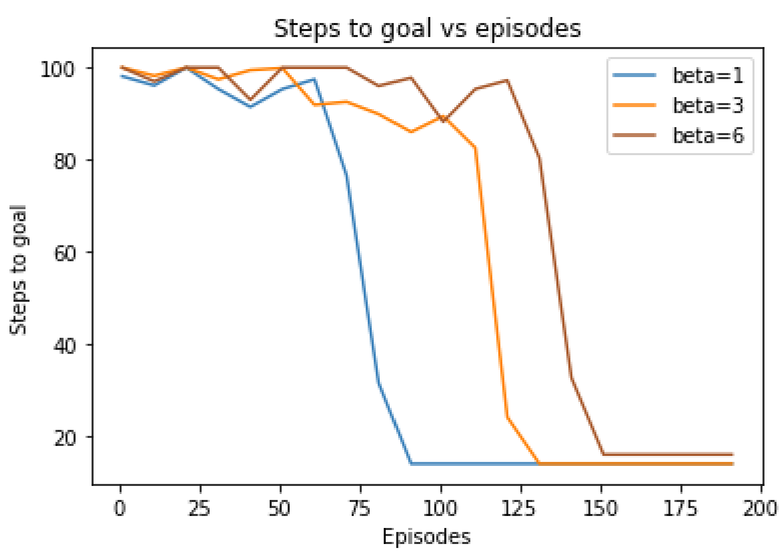
Comments

Here we see that exploration rate of 0.01 and 0.1 are very similar in terms of convergence speed an number of steps it took to goals. Note that exploration rate of 0.05 takes approximately 40 steps to goals. This is because the chance of randomly choose an action instead of optimal action is 50%. Hence the model is likely to make suboptimal path choosing - taking longer to goals. Though by some miracle or unknowing luck, the model may choose optimal path with 50% rate. Hence everytime you run this model you will get a different plot.

(b) Try exploring with policy derived from **softmax of Q-values** described in the Q learning lecture. Use the values of $\beta \in \{1, 3, 6\}$ for your experiment, keeping β fixed throughout the training.

In [123]:
```
# TODO: Fill this in for Static Beta with softmax of Q-values
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100


# TODO: Set the beta
beta_list = [1, 3, 6]
use_softmax_policy = True
k_exp_schedule = 0.1 # (float) choose k such that we have a constant beta duri
ng training

env = MazeEnv()
steps_vs_iters_list = []
for beta in beta_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, max_
steps, use_softmax_policy, beta, k_exp_schedule)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [124]:
```
label_list = ["beta={}".format(beta) for beta in beta_list]
# TODO:
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```



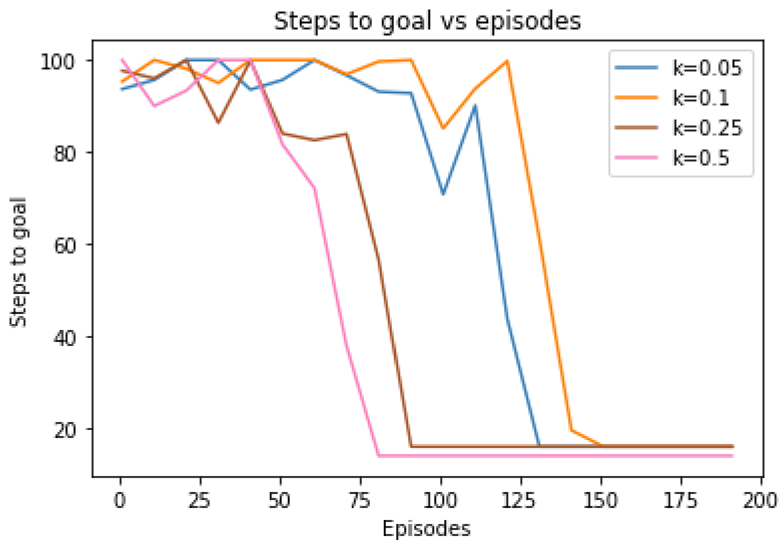
(c) Instead of fixing the $\beta = \beta_0$ to the initial value, we will increase the value of β as the number of episodes t increase:

$$\beta(t) = \beta_0 e^{kt}$$

That is, the β value is fixed for a particular episode. Run the training again for different values of $k \in \{0.05, 0.1, 0.25, 0.5\}$, keeping $\beta_0 = 1.0$. Compare the results obtained with this approach to those obtained with a static β value.

In [132]:
```
# TODO: Fill this in for Dynamic Beta
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100


# TODO: Set the beta
beta = 1.0
use_softmax_policy = True
k_exp_schedule_list = [0.05, 0.1, 0.25, 0.5]
env = MazeEnv()

steps_vs_iters_list = []
for k_exp_schedule in k_exp_schedule_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, max_
steps, use_softmax_policy, beta, k_exp_schedule)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [133]:
```
# TODO: Plot the steps vs iterations
label_list = ["k={}".format(k_exp_schedule) for k_exp_schedule in k_exp_schedu
le_list]
# plot_several_steps_vs_iters(...)

plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```



Comments

As k value increases we can see that B(t) value actually increases exponential (considering the episode = t increases).

$$\beta(t) = \beta_0 e^{kt}$$

So when we do softmax on q_hat*beta, the probability distribution becomes more drastic to concentrate more on the highest probability.

3. Stochastic Environments

(a) Make the environment stochastic (uncertain), such that the agent only has a 95% chance of moving in the chosen direction, and has a 5% chance of moving in some random direction.

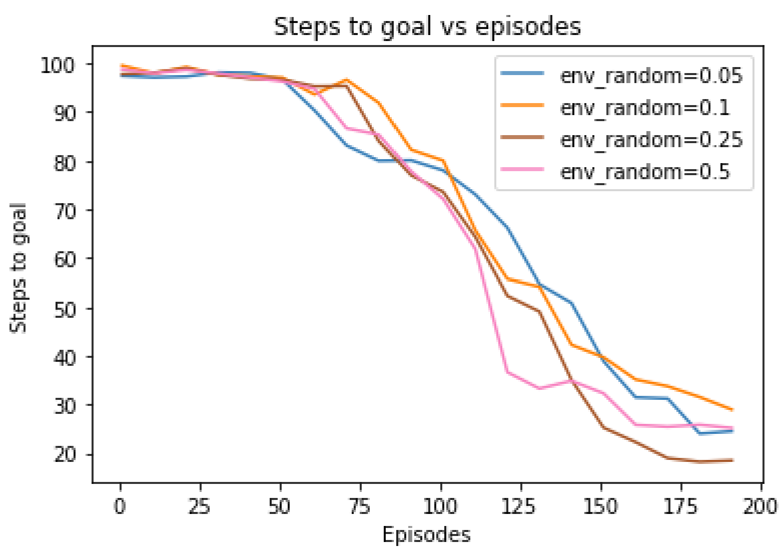
In [54]:
```
# TODO: Implement ProbabilisticMazeEnv in maze.py
```

(b) Change the learning rule to handle the non-determinism, and experiment with different probability of environment performing random action $p_{rand} \in \{0.05, 0.1, 0.25, 0.5\}$ in this new rule. How does performance vary as the environment becomes more stochastic?

Use the same parameters as in first part, except change the alpha (α) value to be **less than 1**, e.g. 0.5.

In [88]:
```
# TODO: Use the same parameters as in the first part, except change alpha
num_iters = 200
alpha = 0.5
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# Set the environment probability of random
env_p_rand_list = [0.05, 0.1, 0.25, 0.5]

steps_vs_iters_list = []
for env_p_rand in env_p_rand_list:
    # Instantiate with ProbabilisticMazeEnv
    env = MazeEnv()
    p_env = ProbabilisticMazeEnv(env, env_p_rand)

    # Note: We will repeat for several runs of the algorithm to make the resul
t less noisy
    avg_steps_vs_iters = np.zeros(num_iters)
    for i in range(10):
        q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon,
max_steps, use_softmax_policy)
        avg_steps_vs_iters += steps_vs_iters
    avg_steps_vs_iters /= 10
    steps_vs_iters_list.append(avg_steps_vs_iters)
```

In [89]:
```
label_list = ["env_random={}".format(env_p_rand) for env_p_rand in env_p_rand_
list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```



Comments

ProbabilisticMaze is very similar to how epsilon-greedy works. We're basically imposing another [5%, 10%, 25%, 50%] uncertainty in choosing a random direction (after we've selected optimal action using e-greedy). Hence the model has more variace in that it will be more 'exploratory' given the extra percentages. Notice in the graph that 4 models basically need more episodes to get to the goal. In addition, they took more steps to the goals than our previous epsilon-greedy model. The environment becomes more stochastic and is trying to explore its path (could be swerving out of optimal path). Hence taking longer to converge and longer to goal.

3. Did you complete the course evaluation?

In [151]:
```
# Answer: yes / no
```

Yes

Supplemental code

Maze.py

In [ ]:
```
import numpy as np
import copy
import math

ACTION_MEANING = {
    0: "UP",
    1: "RIGHT",
    2: "LEFT",
    3: "DOWN",
}

SPACE_MEANING = {
    1: "ROAD",
    0: "BARRIER",
    -1: "GOAL",
}


class MazeEnv:

    def __init__(self, start=[6,3], goals=[[1, 8]]):
        """Deterministic Maze Environment"""

        self.m_size = 10
        self.reward = 10
        self.num_actions = 4
        self.num_states = self.m_size * self.m_size

        self.map = np.ones((self.m_size, self.m_size))
        self.map[3, 4:9] = 0
        self.map[4:8, 4] = 0
        self.map[5, 2:4] = 0

        for goal in goals:
            self.map[goal[0], goal[1]] = -1

        self.start = start
        self.goals = goals
        self.obs = self.start

    def step(self, a):
        """ Perform a action on the environment

            Args:
                a (int): action integer

            Returns:
                obs (list): observation list
                reward (int): reward for such action
                done (int): whether the goal is reached
        """
        done, reward = False, 0.0
        next_obs = copy.copy(self.obs)

        if a == 0:
            next_obs[0] = next_obs[0] - 1
```

```
        elif a == 1:
            next_obs[1] = next_obs[1] + 1
        elif a == 2:
            next_obs[1] = next_obs[1] - 1
        elif a == 3:
            next_obs[0] = next_obs[0] + 1
        else:
            raise Exception("Action is Not Valid")

        if self.is_valid_obs(next_obs):
            self.obs = next_obs

        if self.map[self.obs[0], self.obs[1]] == -1:
            reward = self.reward
            done = True

        state = self.get_state_from_coords(self.obs[0], self.obs[1])

        return state, reward, done

    def is_valid_obs(self, obs):
        """ Check whether the observation is valid

            Args:
                obs (list): observation [x, y]

            Returns:
                is_valid (bool)
        """

        if obs[0] >= self.m_size or obs[0] < 0:
            return False

        if obs[1] >= self.m_size or obs[1] < 0:
            return False

        if self.map[obs[0], obs[1]] == 0:
            return False

        return True

    @property
    def _get_obs(self):
        """ Get current observation
        """
        return self.obs

    @property
    def _get_state(self):
        """ Get current observation
        """
        return self.get_state_from_coords(self.obs[0], self.obs[1])

    @property
    def _get_start_state(self):
        """ Get the start state
        """
```

```
        return self.get_state_from_coords(self.start[0], self.start[1])

    @property
    def _get_goal_state(self):
        """ Get the start state
        """
        goals = []
        for goal in self.goals:
            goals.append(self.get_state_from_coords(goal[0], goal[1]))
        return goals

    def reset(self):
        """ Reset the observation into starting point
        """
        self.obs = self.start
        state = self.get_state_from_coords(self.obs[0], self.obs[1])
        return state

    def get_state_from_coords(self, row, col):
        state = row * self.m_size + col
        return state

    def get_coords_from_state(self, state):
        row = math.floor(state/self.m_size)
        col = state % self.m_size
        return row, col


class ProbabilisticMazeEnv(MazeEnv):
    """ (Q2.3) Hints: you can refer the implementation in MazeEnv
    """

    def __init__(self, goals=[[2, 8]], p_random=0.05):
        """ Probabilistic Maze Environment

            Args:
                goals (list): list of goals coordinates
                p_random (float): random action rate
        """
        MazeEnv.__init__(self)
        self.p_random = p_random


    def step(self, a):


        done, reward = False, 0.0
        next_obs = copy.copy(self.obs)

        # Create variable with uniform[0,1] distribution
        sample = np.random.uniform()

        if sample < self.p_random:
            a = np.random.randint(4)


        if a == 0:
```

```
            next_obs[0] = next_obs[0] - 1
        elif a == 1:
            next_obs[1] = next_obs[1] + 1
        elif a == 2:
            next_obs[1] = next_obs[1] - 1
        elif a == 3:
            next_obs[0] = next_obs[0] + 1
        else:
            raise Exception("Action is Not Valid")

        if self.is_valid_obs(next_obs):
            self.obs = next_obs

        if self.map[self.obs[0], self.obs[1]] == -1:
            reward = self.reward
            done = True

        state = self.get_state_from_coords(self.obs[0], self.obs[1])

        return state, reward, done
```

qlearning.py

In [ ]:
```
import numpy as np
import math
import copy

def qlearn(env, num_iters, alpha, gamma, epsilon, max_steps, use_softmax_polic
y, init_beta=None, k_exp_sched=None):
    """ Runs tabular Q learning algorithm for stochastic environment.

    Args:
        env: instance of environment object
        num_iters (int): Number of episodes to run Q-learning algorithm
        alpha (float): The learning rate between [0,1]
        gamma (float): Discount factor, between [0,1)
        epsilon (float): Probability in [0,1] that the agent selects a random
 move instead of
                selecting greedily from Q value
        max_steps (int): Maximum number of steps in the environment per episod
e
        use_softmax_policy (bool): Whether to use softmax policy (True) or Eps
ilon-Greedy (False)
        init_beta (float): If using stochastic policy, sets the initial beta a
s the parameter for the softmax
        k_exp_sched (float): If using stochastic policy, sets hyperparameter f
or exponential schedule
            on beta

    Returns:
        q_hat: A Q-value table shaped [num_states, num_actions] for environmen
t with with num_states
            number of states (e.g. num rows * num columns for grid) and num_ac
tions number of possible
            actions (e.g. 4 actions up/down/left/right)
        steps_vs_iters: An array of size num_iters. Each element denotes the n
umber
            of steps in the environment that the agent took to get to the goal
            (capped to max_steps)
    """

    action_space_size = env.num_actions
    state_space_size = env.num_states
    q_hat = np.zeros(shape=(state_space_size, action_space_size))
    steps_vs_iters = np.zeros(num_iters)


    for i in range(num_iters):
        # TODO: Initialize current state by resetting the environment
        # curr_state = ...
        curr_state = env.reset()
        num_steps = 0
        done = False

        # TODO: Keep looping while environment isn't done and less than maximu
m steps
        while done == False and num_steps < max_steps:
            num_steps += 1
            # Choose an action using policy derived from either softmax Q-valu
```

```
e
            # or epsilon greedy
            if use_softmax_policy:
                assert(init_beta is not None)
                assert(k_exp_sched is not None)
                # TODO: Boltzmann stochastic policy (softmax policy)
                # beta = ... # Call beta_exp_schedule to get the current beta
 value
                beta = beta_exp_schedule(init_beta, i, k_exp_sched)
                # action = ...
                action = softmax_policy(q_hat, beta, curr_state)
            else:
                # TODO: Epsilon-greedy
                # action = ...
                action = epsilon_greedy(q_hat, epsilon, curr_state, action_spa
ce_size)

            # TODO: Execute action in the environment and observe the next sta
te, reward, and done flag
            # next_state, reward, done = ...

            next_state, reward, done = env.step(action)

            # TODO: Update Q_value
            if next_state != curr_state:
                new_value = q_hat[curr_state][action]

                # TODO: Use Q-learning rule to update q_hat for the curr_state
and action:
                # i.e., Q(s,a) <- Q(s,a) + alpha*[reward + gamma * max_a'(Q
(s',a')) - Q(s,a)]
                q_hat[curr_state][action] = (1 - alpha)*new_value + \
                        alpha*(reward + gamma*np.max(q_hat[next_state]))

                # TODO: Update the current state to be the next state
                curr_state = next_state

        steps_vs_iters[i] = num_steps

    return q_hat, steps_vs_iters


def epsilon_greedy(q_hat, epsilon, state, action_space_size):
    """ Chooses a random action with p_rand_move probability,
    otherwise choose the action with highest Q value for
    current observation

    Args:
        q_hat: A Q-value table shaped [num_rows, num_col, num_actions] for
            grid environment with num_rows rows and num_col columns and num_ac
tions
            number of possible actions
        epsilon (float): Probability in [0,1] that the agent selects a random
            move instead of selecting greedily from Q value
        state: A 2-element array with integer element denoting the row and col
umn
            that the agent is in
```

```
        action_space_size (int): number of possible actions

    Returns:
        action (int): A number in the range [0, action_space_size-1]
            denoting the action the agent will take
    """
    # TODO: Implement your code here
    # Hint: Sample from a uniform distribution and check if the sample is belo
w
    # a certain threshold


    sample = np.random.uniform(0,1)


    if np.all(q_hat[state]==0):
        return np.random.randint(0, action_space_size)

    elif sample > epsilon:
        return np.argmax(q_hat[state])

    else:
        return np.random.randint(0, action_space_size)



def softmax_policy(q_hat, beta, state):
    """ Choose action using policy derived from Q, using
    softmax of the Q values divided by the temperature.

    Args:
        q_hat: A Q-value table shaped [num_rows, num_col, num_actions] for
            grid environment with num_rows rows and num_col columns
        beta (float): Parameter for controlling the stochasticity of the actio
n
        state: A 2-element array with integer element denoting the row and col
umn
            that the agent is in

    Returns:
        action (int): A number in the range [0, action_space_size-1]
            denoting the action the agent will take
    """
    # TODO: Implement your code here
    # Hint: use the stable_softmax function defined below
    # Multiply Q values by beta
    Bq_hat = q_hat*beta

    if np.all(Bq_hat[state]==0):
        return np.random.randint(4)
    # Compute the softmax probability
    softmax = stable_softmax(Bq_hat)

    # Get the action using argmax at the current state
    action = np.argmax(softmax[state])

    return action
```

```python
def beta_exp_schedule(init_beta, iteration, k=0.1):
    beta = init_beta * np.exp(k * iteration)
    return beta

def stable_softmax(x, axis=1):
    """ Numerically stable softmax:
    softmax(x) = e^x /(sum(e^x))
               = e^x / (e^max(x) * sum(e^x/e^max(x)))

    Args:
        x: An N-dimensional array of floats
        axis: The axis for normalizing over.

    Returns:
        output: softmax(x) along the specified dimension
    """
    max_x = np.max(x, axis, keepdims=True)
    z = np.exp(x - max_x)
    output = z / np.sum(z, axis, keepdims=True)

    return output
```