Spring 2021 6.884 Computational Sensorimotor Learning Assignment 2

In this assignment, we will implement three principle reinforcement learning algorithms which provably converge to optimal solutions for MDPs:

- Value iteration
- Policy iteration
- Q-learning

You will need to answer the bolded questions and fill in the missing code snippets (marked by **TODO**).

There are (approximately) **239** total points to be had in this PSET. `ctrl-f` for "pts" to ensure you don't miss questions.

Please fill in your name below:

Name: Songhao Li

Credits

Some part of the code of this assignment is borrowed from the Spring 2018 CMU Deep Reinforcement Learning & Control course. We also thank Prof. Cathy Wu (https://idss.mit.edu/staff/cathy-wu/) for polishing the content.

Setup

The following code sets up imports and helper functions (you can ignore this).

In [ ]:
```
%matplotlib inline
import numpy as np
import random
import time
import os
import gym
import json
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
import pandas as pd


from copy import deepcopy
from tqdm.notebook import tqdm
from dataclasses import dataclass
from matplotlib import animation
from IPython.display import HTML
from typing import Any
from collections import deque

mpl.rcParams['figure.dpi']= 100
```

In [ ]:
```
# some util functions
def plot(logs, x_key, y_key, legend_key, **kwargs):
    nums = len(logs[legend_key].unique())
    palette = sns.color_palette("hls", nums)
    if 'palette' not in kwargs:
        kwargs['palette'] = palette
    sns.lineplot(x=x_key, y=y_key, data=logs, hue=legend_key, **kwargs)

def set_random_seed(seed):
    np.random.seed(seed)
    random.seed(seed)

# set random seed
seed = 0
set_random_seed(seed=seed)
```

FrozenLake environment

Winter has come. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend.

The surface is described using a grid like the following:

```
SFFF # (S: starting point, safe)
FHFH # (F: frozen surface, safe)
FFFH # (H: hole, fall to your doom)
HFFG # (G: goal, where the frisbee is located)
```

The episode ends when you reach the goal or fall in a hole. You receive a reward of 1 if you reach the goal, and zero otherwise.

Here's what the Frozen Lake looks like in action, when following a random agent:

Frozen Lake is part of OpenAI gym, a collection of open-source environments for benchmarking RL algorithms. [Here (https://gym.openai.com/envs/FrozenLake-v0/)](https://gym.openai.com/envs/FrozenLake-v0/) is a link to the gym environment (also the source of our environment description).

Question: how many actions can the agent take at any time; eg, what is the size of the action space? (2 pts)

Answer: The agent has four actions to take at any time, since it can decide to go up, down, left, or right.

Now, here's some code that creates the above environment through OpenAI gym, called `FrozenLake-v0` . Note that we will be using a deterministic variant, so providing an action in a given direction will always move you in that direction.

In [ ]:
```
## create FrozenLake environment
MAPS = {
    "4x4": [
        "GHFS",
        "FHHF",
        "FFHF",
        "FFFF"
    ],
    "8x8": [
        "FFFSFFFF",
        "FFFFFFFF",
        "HHHHFHFF",
        "FFFFFFHF",
        "FFFFFFFF",
        "FHFFFFHF",
        "FHFFHFHH",
        "FGHFFFFF"
    ],
}
from gym.envs.registration import register
env_name = 'Deterministic-4x4-FrozenLake-v0'
if env_name not in gym.envs.registry.env_specs:
    register(
        id=env_name,
        entry_point='gym.envs.toy_text.frozen_lake:FrozenLakeEnv',
        kwargs={'map_name': None,
                'is_slippery': False,
                'desc': MAPS['4x4']
                },
        max_episode_steps=20)

env_name = 'Deterministic-8x8-FrozenLake-v0'
if env_name not in gym.envs.registry.env_specs:
    register(
        id=env_name,
        entry_point='gym.envs.toy_text.frozen_lake:FrozenLakeEnv',
        kwargs={'map_name': None,
                'is_slippery': False,
                'desc': MAPS['8x8'],
                },
        max_episode_steps=100)
```

We would like to find a good policy for the agent (you, the brave soul). More precisely, the agent controls the movement of a character on the frozen lake. The frozen lake environment is an example of a *grid world*, since it consists of objects moving around in a discrete (grid) world. Grid world problems can constitute or approximate a wide class of problems.

Question: Consider a racecar environment, where the goal is to get the agent (the racecar) around a race track as quickly as possible. Is this suitable for representing as a grid world problem? Justify your answer. (4 pts)

Answer: Yes, sine the race track is a discrete environment.

Now let's return to the task at hand: retrieving our frisbee! Remember that some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

Now let's consider some algorithms for solving this problem, i.e. finding a good policy to accomplish the task.

Value Iteration

Value iteration is the first algorithm we will consider.

Let's first do a sanity check. In class, we learned that value iteration is model-based and as such, it is best for problems with small state spaces.

Question: Consider a fixed 4x4-grid FrozenLake. What is the size of the state space? (4 pts)

Answer: 16

Now recall from class that value iteration is a model-based method which starts with any guess of a value function V_0 and then updates it according to the optimal Bellman equation: $V_{i+1}(s) = \mathcal{T}V_i(s) = \max_a \mathbb{E}_{s' \sim p(\cdot|s,a)}[r(s,a,s') + \gamma V_i(s')]$. In our case, as we have a fixed number of states, our value function is simply a mapping of square -> value (eg an array).

Note: we choose to represent actions as integers with the following mapping:

```
LEFT = 0
DOWN = 1
RIGHT = 2
UP = 3
```

and states as zero-indexed integers traversing from the top left.

Implement Value Iteration

(20 pts)

We will now implement value iteration over the MDP with transition probabilities described by `env.P` (transition probabilites), `env.nS` (number of states), and `env.nA` (number of actions). The entry `env.P[s][a]` (where `s` is the state index and `a` is the action index) is a list of transition tuples $(p(s,a,s'), s', r(s,a,s'), \text{episode\_end})$ for each list index index `s'`. In plain english:

- $p(s,a,s')$: the probability of transitioning to state `s'` after taking action `a` in state `s`.
- s': the next state under this transition.
- $r(s,a,s')$: the reward of taking this transition.
- `episode_end`: a boolean representing whether taking this action ends the episode (in Frozen lLke, whether this kills you or takes you to the goal position).

In [ ]:
```
def value_iteration(env, gamma, max_iterations=1000, tol=1e-3):
    """Runs value iteration for a given gamma and environment.
    Updates states in their 1-N order.
    Parameters
    ----------
    env: gym.core.Environment
      The environment to compute value iteration for. Must have nS,
      nA, and P as attributes.
    gamma: float
      Discount factor, must be in range [0, 1)
    max_iterations: int
      The maximum number of iterations to run before stopping.
    tol: float
      Determines when value function has converged.
    Returns
    -------
    np.ndarray, iteration, list
      The value function, the number of iterations it took to converge, and a
 list
      of the value functions after each iteration.
    """
    value_func = np.zeros(env.nS) # initial value function: all states are zer
o
    policy = np.zeros(env.nS, dtype='int') # placeholder for computed policy
    iters = 0
    value_history = []
    while True:
        delta = 0
        ###### TODO: value function update ##############
        value_func_copy = value_func.copy()
        for state in range(env.nS):
            action_values = np.zeros(env.nA)
            for action in range(env.nA):
                action_value = 0
                for i in range(len(env.P[state][action])):
                    prob, next_state, reward, done = env.P[state][action][i]
                    if done:
                        action_value += prob*reward
                    else:
                        action_value += prob*reward + gamma*value_func_copy[ne
xt_state]
                action_values[action] = action_value
            best_action = np.argmax(action_values)
            best_reward = np.max(action_values)
            delta += abs(value_func[state] - best_reward)
            value_func[state] = best_reward
            policy[state] = best_action
        #################################################################
        # let's also save a copy of value function after each iteration
        value_history.append(value_func.copy())
        iters += 1
        if delta < tol or iters >= max_iterations:
            break
    return value_func, iters, value_history
```

Actually, computing the optimal value function is not enough. What we are interested in is the optimal policy, not just how good each state is. Luckily, the optimal value function and the optimal policy are related. Let's implement this next:

(15 pts)

In [ ]:
```
def value_function_to_policy(env, gamma, value_function):
    """Output action numbers for each state in value_function.
    Parameters
    ----------
    env: gym.core.Environment
      Environment to compute policy for. Must have nS, nA, and P as
      attributes.
    gamma: float
      Discount factor. Number in range [0, 1)
    value_function: np.ndarray
      Value of each state.
    Returns
    -------
    np.ndarray
      An array of integers. Each integer is the optimal action to take
      in that state according to the environment dynamics and the
      given value function.
    """
    policy = np.zeros(env.nS, dtype='int')
    for idx in range(env.nS):
        p = env.P[idx]
        vmax = -np.inf
        best_act = -1
        state = idx
        ###### TODO: Select the best action (best_act) ##############
        for action in range(env.nA):
            action_value = 0
            for next_state in env.P[state][action]:
                action_value += next_state[0]*next_state[2] + gamma*value_func
tion[next_state[1]]
            if action_value > vmax:
                best_act = action
                vmax = action_value
        ##################################################

        policy[idx] = best_act
    return policy
```

Question: What is the difference between the value iteration algorithm and extracting a policy from a value function? (4 pts)

Answer:

OK enough talk, let's run it. We'll consider this 8x8 frozen lake:

```
FFFSFFFF
FFFFFFFF
HHHHFHFF
FFFFFFHF
FFFFFFFF
FHFFFFHF
FHFFHFHH
FGHFFFFF
```

Run Value Iteration

(10 pts)

Print and plot your computed value and policy functions as 8x8 2D grids. Your policy should look something like this:

```
XXXXXXXX
XXXXXXXX
...
XXXXXXXX
```

where `X` is the optimal action in that state (one of `L` , `R` , `U` , `D`). Your value function should look similar, except containing space-separated floats rather than action characters.

For `plot_value_history` , plot each value function in the history using the `plt.imgshow` method from `matplotlib` . To plot multiple images at once, you may find `plt.subplot` useful.

In [ ]:
```
def print_policy(policy):
    ### TODO: implement method ################################
    for i in range(8):
        string = str()
        for j in range(8):
            if policy[8*i+j] == 0:
                string += 'L'
            elif policy[8*i+j] == 1:
                string += 'D'
            elif policy[8*i+j] == 2:
                string += 'R'
            else:
                string += 'U'
        print(string)
    ##########################################################


def print_value(value):
    ### TODO: implement method ################################
    for i in range(8):
        string = str()
        for j in range(8):
            if value[8*i+j] == 0:
                string += '0.000'
            else:
                string += str(value[8*i+j])[:5]
            string += ' '
        print(string)
    return
    ##########################################################

def plot_value_history(value_history):
    ### TODO: implement method ################################
    idx = 0
    for value in value_history:
      idx += 1
      print('number %d' %idx)
      plt.imshow(value.reshape(8,8))
      plt.show()
    ##########################################################
```

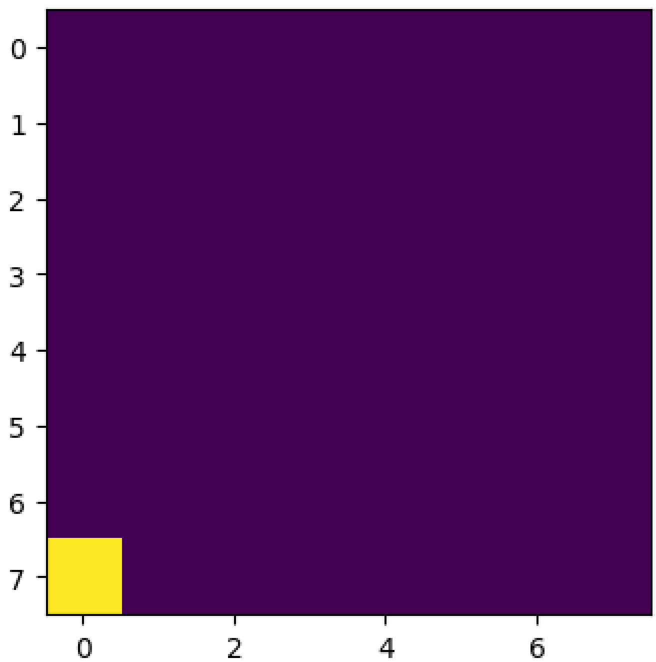
In [ ]:
```
env = gym.make('Deterministic-8x8-FrozenLake-v0')
gamma = 0.9
value_info = value_iteration(env, gamma, max_iterations=int(1e3), tol=1e-4)
value, iters, value_history = value_info
policy = value_function_to_policy(env, gamma, value)
print('Policy: ')
print_policy(policy)
print('Value: ')
print_value(value)
print('Iterations: ', iters)
```

```
Policy:
DDDDDLLL
RRRRDLLL
LLLLDLRD
DLLLLLLD
DLLLLLLL
DLULLLLU
DLULLULL
RLLULLLL
Value:
0.205 0.228 0.254 0.282 0.313 0.282 0.254 0.228
0.228 0.254 0.282 0.313 0.348 0.313 0.282 0.254
0.000 0.000 0.000 0.000 0.387 0.000 0.254 0.282
0.656 0.590 0.531 0.478 0.430 0.387 0.000 0.313
0.729 0.656 0.590 0.531 0.478 0.430 0.387 0.348
0.81 0.000 0.531 0.478 0.430 0.387 0.000 0.313
0.9 0.000 0.478 0.430 0.000 0.348 0.000 0.000
1.0 0.000 0.000 0.387 0.348 0.313 0.282 0.254
Iterations:  17
```

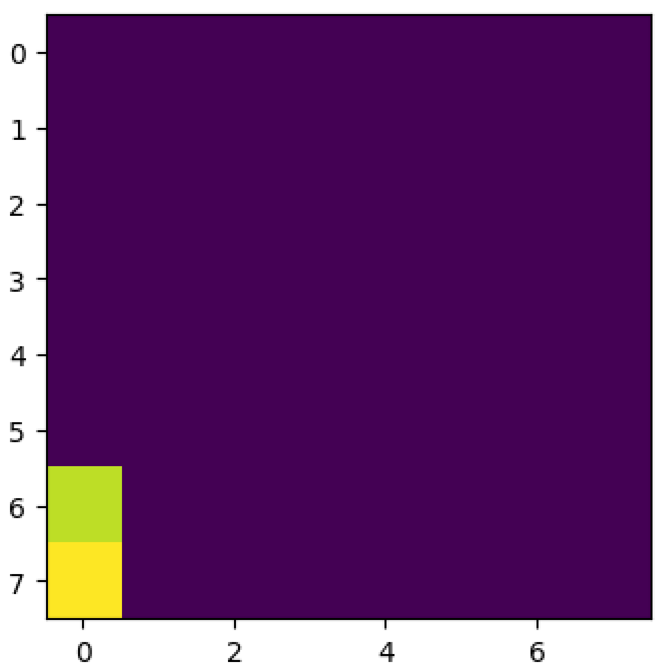
```
FFFSFFFF
FFFFFFFF
HHHHFHFF
FFFFFFHF
FFFFFFFF
FHFFFFHF
FHFFHFHH
FGHFFFFF
```

In [ ]:
```
plot_value_history(value_history)
```

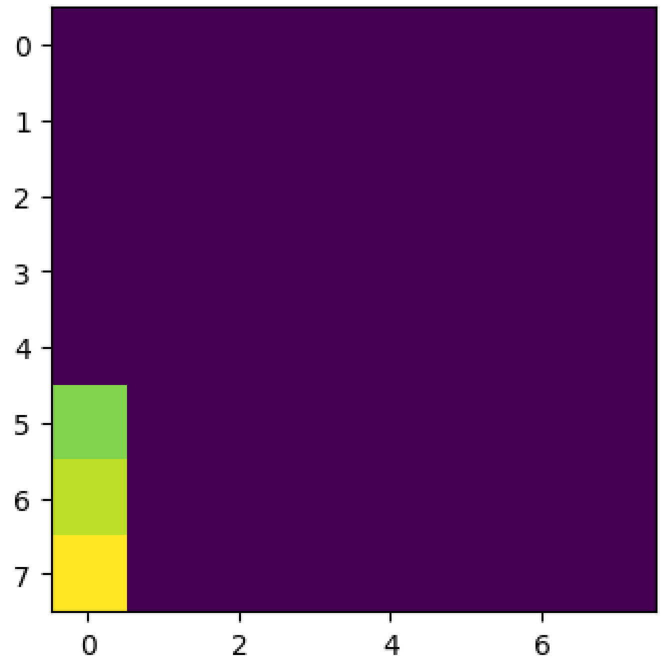
number 1



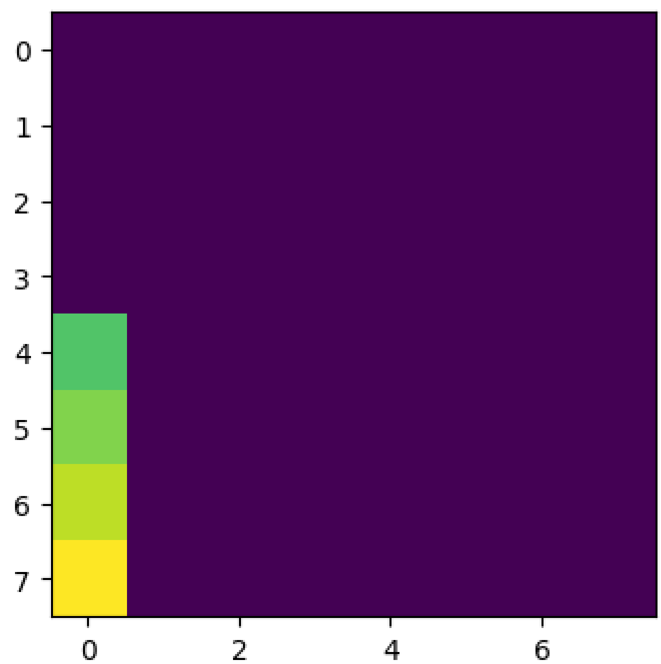
number 2



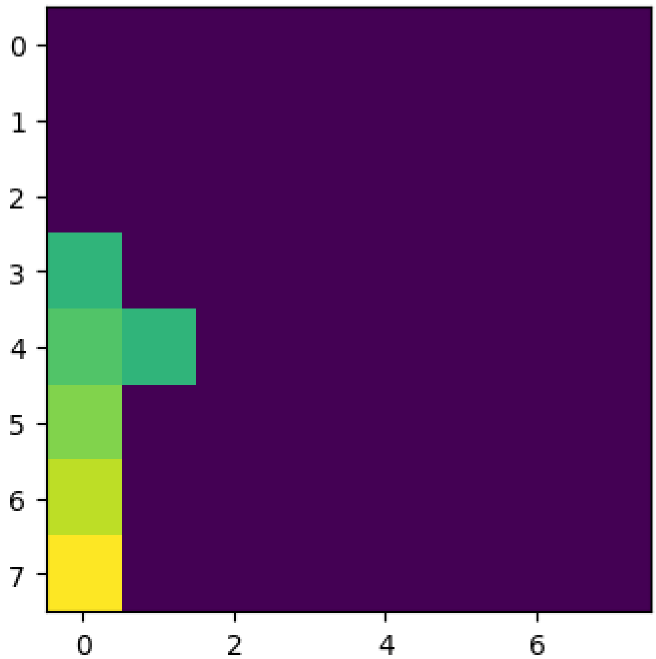
number 3



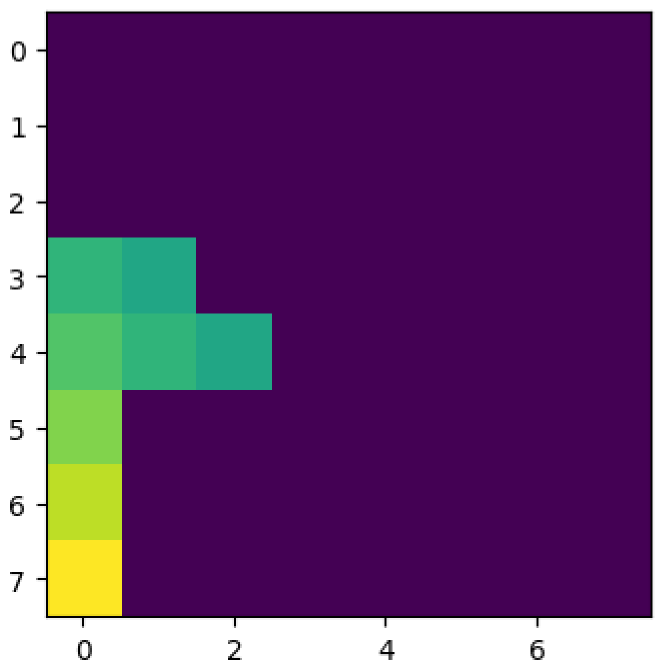
number 4



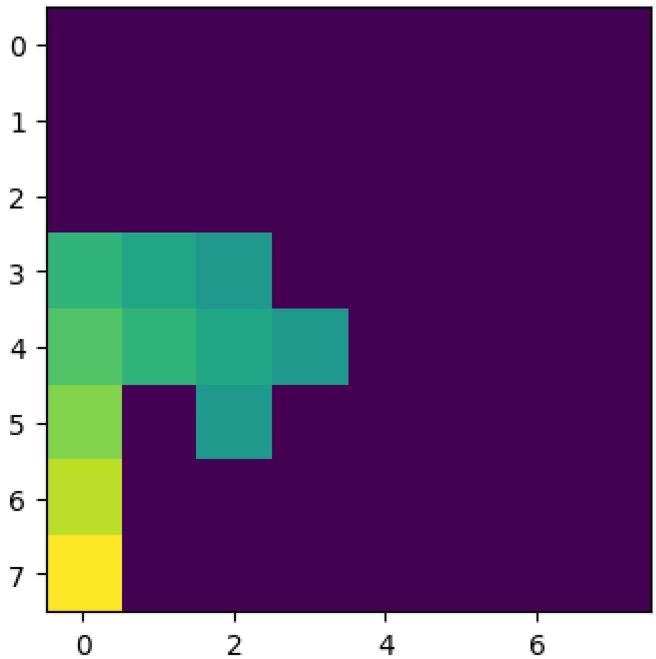
number 5



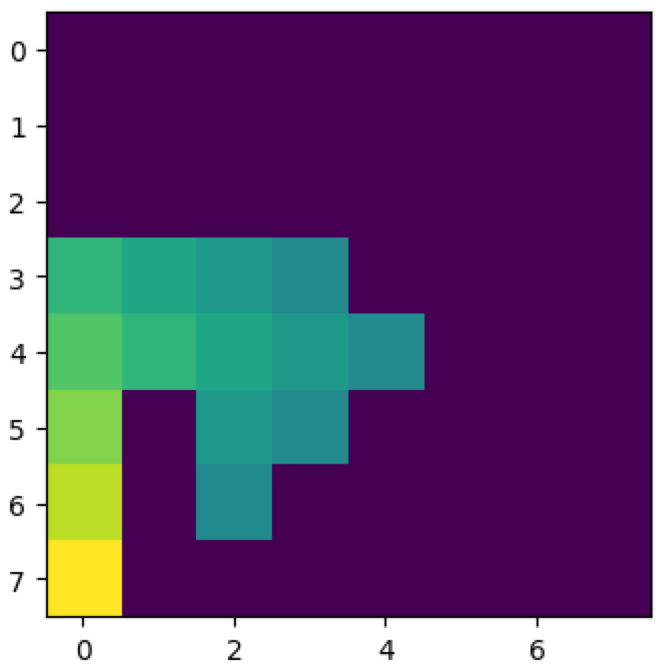
number 6



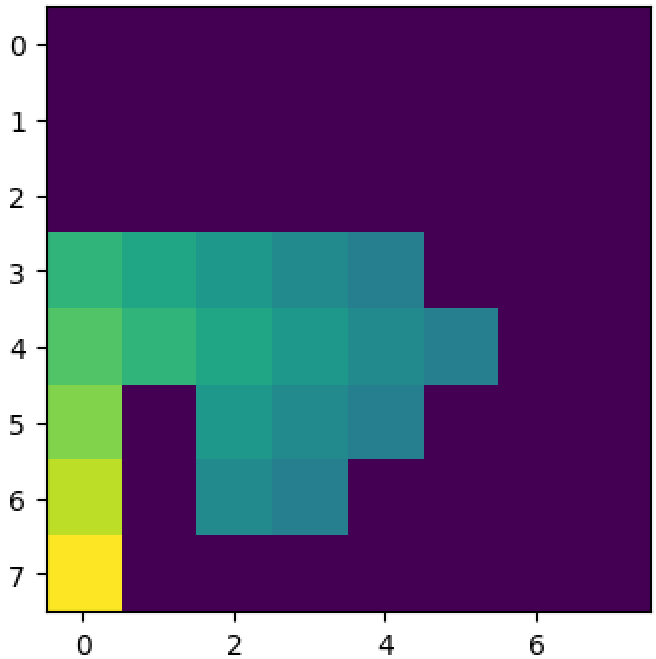
number 7



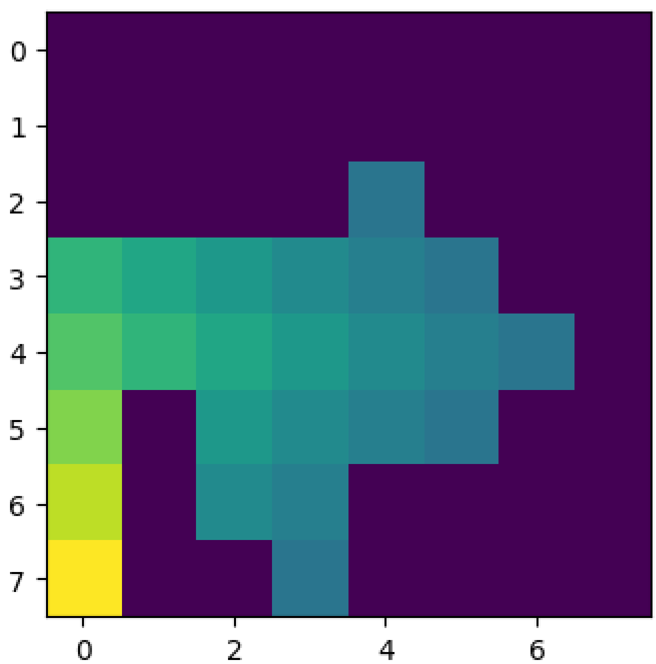
number 8



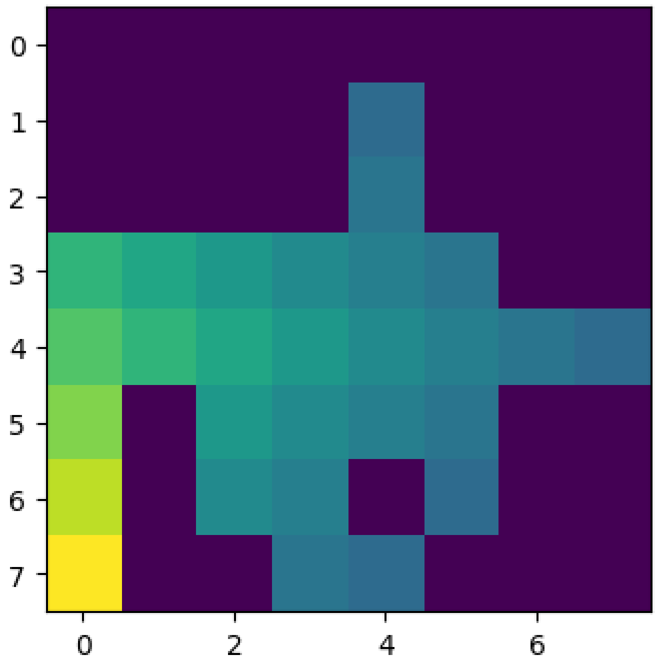
number 9



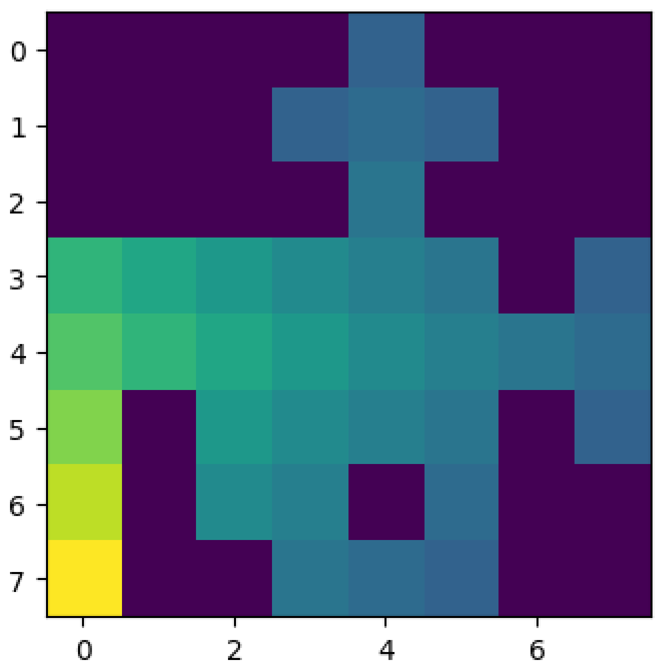
number 10



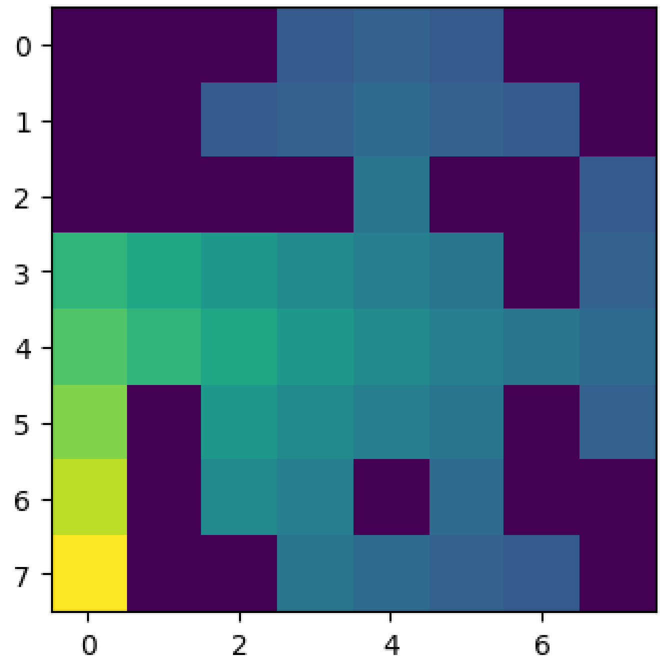
number 11



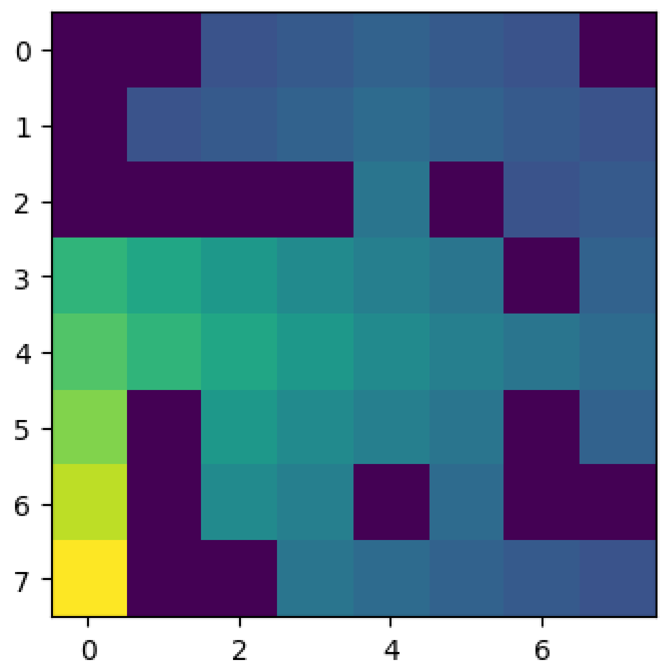
number 12



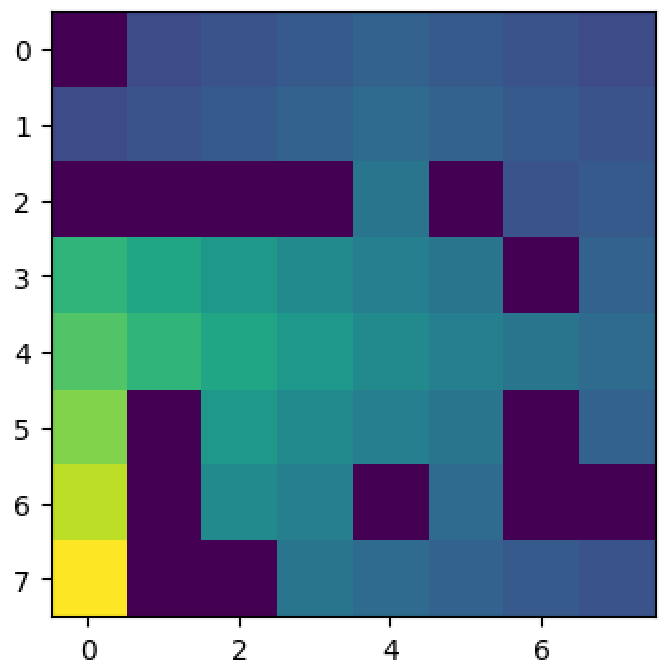
number 13



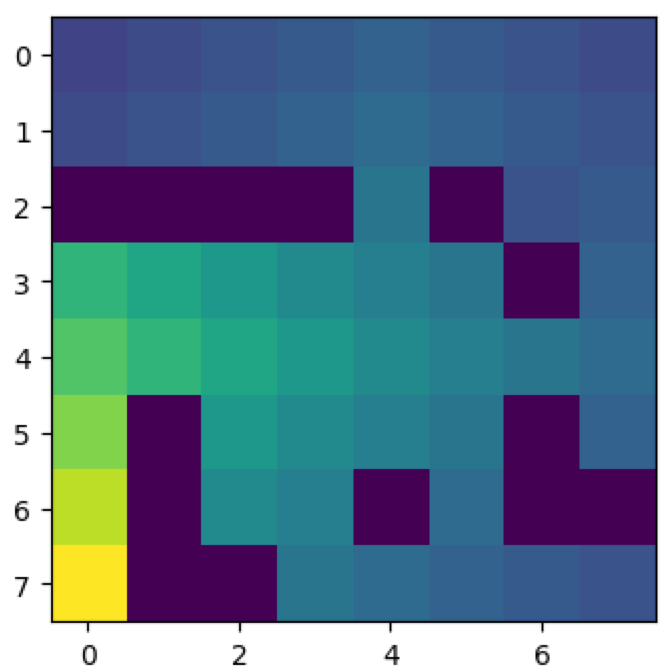
number 14



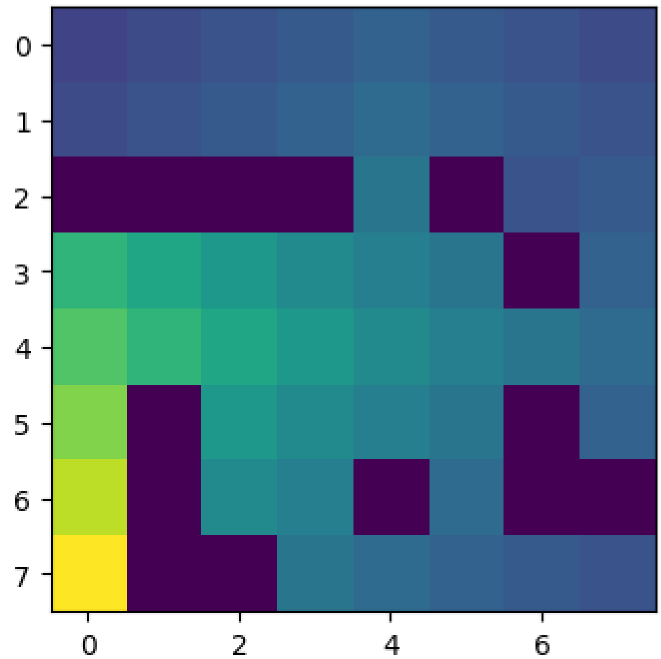
number 15



number 16



number 17



Question: Answers/observe the following *(18 pts)*:

- Has the agent found a successful policy? (hint: it should have.) How do you know?
  - **Answer**: It did found a successful policy, since the number of iteration is less than max_iteration, which means the policy converged.
- Does the value function seem reasonable? Please explain.
  - **Answer**: It is reasonable since: 1) all values are below 1; 2)all values for holes are 0; 3) the value of the state left to the goal is 1
- How many steps of value iteration were required convergence?
  - **Answer**: Here the method I implemented took 17 steps. However, if the values are updated in the process of an iteration, it will be 13 steps.
- How many steps does the agent take to reach the goal location?
  - **Answer**: 16 steps by looking at printed policy

- Try running the above code with gamma = 0.3. Does the agent converge to a successful policy (yes/no)?
  - **Answer**: No
- If yes, then why does gamma have no effect? If no, then why does gamma matter?
  - **Answer**: Because when gamma is small, future states means less and less unsignificant and the value table is "short sighted". Techically speaking, when gamma is small enough some updates are smaller than the convergence shreshold, and the iteration stops right there.

Policy Iteration

Recall from class that value iteration is a special instance of generalized policy iteration, which alternates between 1 step of policy evaluation and 1 step of policy improvement.

Now we'll consider policy iteration, which alternates between many steps of policy evaluation and 1 step of policy improvement. Does that feel silly?

We will make a few notes:

- Let's consider N steps of policy evaluation + 1 step of policy improvement to be 1 iteration of value/policy iteration.
- It is not well understood why, but policy iteration and value iteration will attain the optimal policy in fewer iterations for different problems.
- While policy iteration has a hidden cost of N policy evaluation steps, it turns out that a full policy evaluation can be computed efficiently, since it is a linear operation. (We will not do this in this assignment, but trust us.) If employed efficiently, policy iteration can be viewed as a super-powered value iteration, with accurate policy evaluation and without a whole lot of extra computational cost.

Now, let's implement policy evaluation and policy improvement.

Implement Policy Iteration

(15 pts)

First, implement the the high level wrapper `policy_iteration`, which evaluates a policy to obtain a `value_func` and feeds that into `improve_policy` to update the policy in a loop. Remember to end iteration once the policy is stable.

In [ ]:
```
def policy_iteration(env, gamma, max_iterations=int(1e3), tol=1e-3):
    """Runs policy iteration using the improve_policy and evaluate_policy meth
ods.

    Parameters
    ----------
    env: gym.core.Environment
      The environment to compute value iteration for. Must have nS,
      nA, and P as attributes.
    gamma: float
      Discount factor, must be in range [0, 1)
    max_iterations: int
      The maximum number of iterations to run before stopping.
    tol: float
      Determines when value function has converged.
    Returns
    -------
    (np.ndarray, np.ndarray, int, int, list)
       Returns optimal policy, value function, number of policy
       improvement iterations, number of value iterations, and a list
       of the history value functions.
    """
    policy = np.zeros(env.nS, dtype='int')
    value_history = [] # should contain the value func after each policy itera
tion.
    policy_imp_step = 0 # number of total policy iterations
    policy_eval_step = 0 # number of total value iterations
    while True:
        ### TODO: Fill in policy iteration main loop. ##########
        value_func,iter = evaluate_policy(env,gamma,policy,max_iterations,tol)
        policy_eval_step += iter
        done, new_policy = improve_policy(env, gamma, value_func, policy)
        if done:
            break
        else:
            policy_imp_step += 1
            policy = new_policy
            value_history.append(value_func)
        ##########################################################
    return policy, value_func, policy_imp_step, policy_eval_step, value_histor
y
```

Next, implement the policy evaluation and update submethods.

(25 pts)

In [ ]:
```
def evaluate_policy(env, gamma, policy, max_iterations=int(1e3), tol=1e-3):
    """Performs policy evaluation.
    Evaluates the value of a given policy by asynchronous DP.  Updates states
 in
    their 1-N order.
    Parameters
    ----------
    env: gym.core.Environment
      The environment to compute value iteration for. Must have nS,
      nA, and P as attributes.
    gamma: float
      Discount factor, must be in range [0, 1)
    policy: np.array
      The policy to evaluate. Maps states to actions.
    max_iterations: int
      The maximum number of iterations to run before stopping.
    tol: float
      Determines when value function has converged.
    Returns
    -------
    np.ndarray, int
      The value for the given policy and the number of iterations till
      the value function converged.
    """
    value_func = np.zeros(env.nS)

    iter = 0
    while True:
        delta = 0
        ###### TODO: value function update (value_func) ##############
        new_value_func = np.zeros(env.nS)
        for state in range(env.nS):
            action = policy[state]
            for i in range(len(env.P[state][action])):
                prob, next_state, reward, done = env.P[state][action][i]
                if done:
                    new_value_func[state] = prob*reward
                else:
                    new_value_func[state] = prob*reward + gamma * value_func[n
ext_state]

        delta = np.abs(new_value_func - value_func).sum()
        value_func = new_value_func
        ####################################################
        iter += 1
        if delta < tol or iter >= max_iterations:
            break
    return value_func, iter

def improve_policy(env, gamma, value_func, policy):
    """Performs policy improvement.
    Given a policy and value function, improves the policy.
    Parameters
    ----------
    env: gym.core.Environment
      The environment to compute value iteration for. Must have nS,
```

```
      nA, and P as attributes.
    gamma: float
      Discount factor, must be in range [0, 1)
    value_func: np.ndarray
      Value function for the given policy.
    policy: dict or np.array
      The policy to improve. Maps states to actions.
    Returns
    -------
    bool, np.ndarray
      Returns true if policy changed. Also returns the new policy.
    """
    policy_stable = True
    new_policy = np.zeros(env.nS, dtype='int')
    for idx in range(env.nS):
        old_action = policy[idx]
        p = env.P[idx]
        new_action = -1
        best_q = -np.inf
        ###### TODO: use value function to get new action (new_action) ######
        state = idx
        for action in range(env.nA):
            action_value = 0
            for i in range(len(env.P[state][action])):
                prob, next_state, reward, done = env.P[state][action][i]
                if done:
                    action_value+=prob*reward
                else:
                    action_value+=prob*reward + gamma*value_func[next_state]
            if action_value > best_q:
                new_action = action
                best_q = action_value
        ###################################################
        new_policy[idx] = new_action
        if new_action != old_action:
            policy_stable = False
    return policy_stable, new_policy
```

In [ ]:
```
env.P[1]
```

Out[ ]:
```
{0: [(1.0, 0, 0.0, False)],
 1: [(1.0, 9, 0.0, False)],
 2: [(1.0, 2, 0.0, False)],
 3: [(1.0, 1, 0.0, False)]}
```

Run Policy Iteration

Assuming your above implementation is correct, you should be able to run the below code to evaluate policy iteration on Frozen Lake.

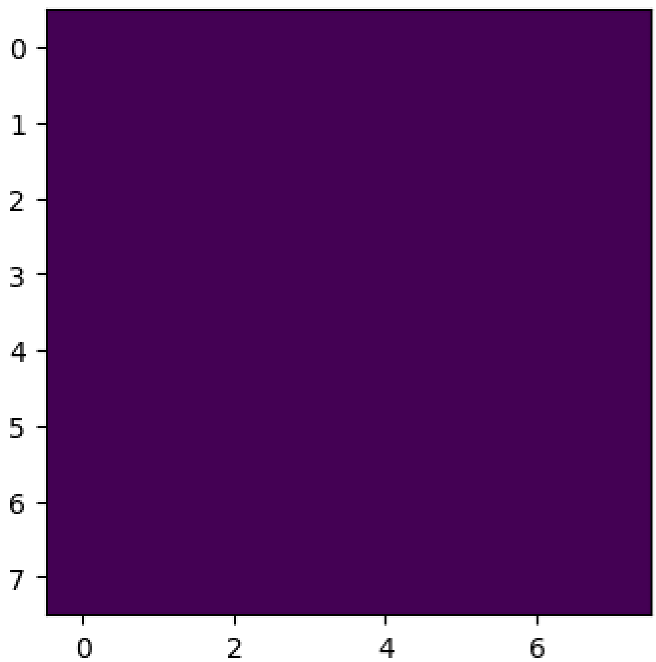
In [ ]:

```
env = gym.make('Deterministic-8x8-FrozenLake-v0')
gamma = 0.9
policy_info = policy_iteration(env, gamma, max_iterations=int(1e3), tol=1e-3)
new_policy, value_func, policy_imp_step, policy_eval_step, value_history = pol
icy_info
policy = value_function_to_policy(env, gamma, value)
print('New policy: ')
print_policy(new_policy)
print('Value: ')
print_value(value_func)
print('Number of policy improvement steps: ', policy_imp_step)
print('Total number of policy evaluation steps: ', policy_eval_step)
```

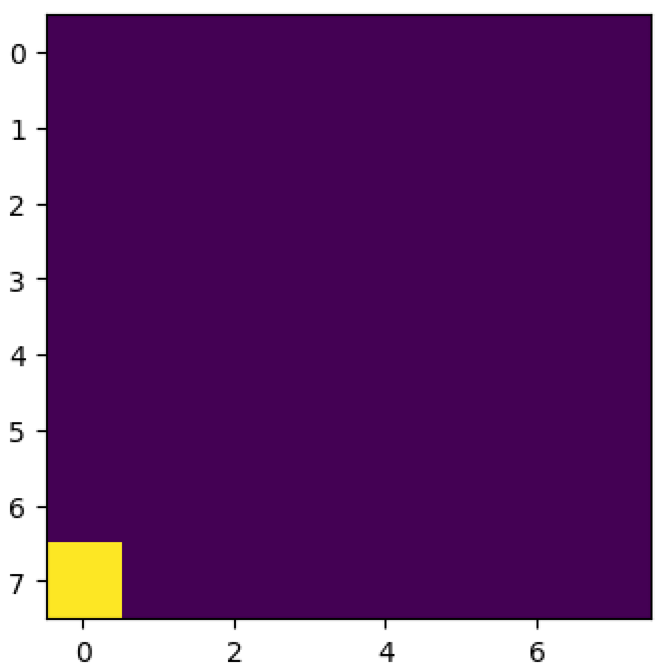
```
New policy:
DDDDDLLL
RRRRDLLL
LLLLDLRD
DLLLLLLD
DLLLLLLL
DLULLLLU
DLULLULL
RLLULLLL
Value:
0.205 0.228 0.254 0.282 0.313 0.282 0.254 0.228
0.228 0.254 0.282 0.313 0.348 0.313 0.282 0.254
0.000 0.000 0.000 0.000 0.387 0.000 0.254 0.282
0.656 0.590 0.531 0.478 0.430 0.387 0.000 0.313
0.729 0.656 0.590 0.531 0.478 0.430 0.387 0.348
0.81 0.000 0.531 0.478 0.430 0.387 0.000 0.313
0.9 0.000 0.478 0.430 0.000 0.348 0.000 0.000
1.0 0.000 0.000 0.387 0.348 0.313 0.282 0.254
Number of policy improvement steps:  12
Total number of policy evaluation steps:  145
```

In [ ]:
```
plot_value_history(value_history)
```

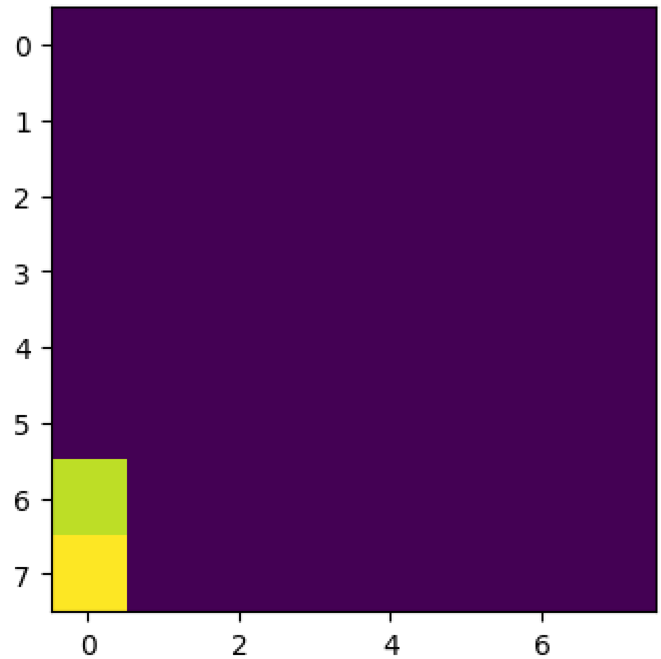
page number 1



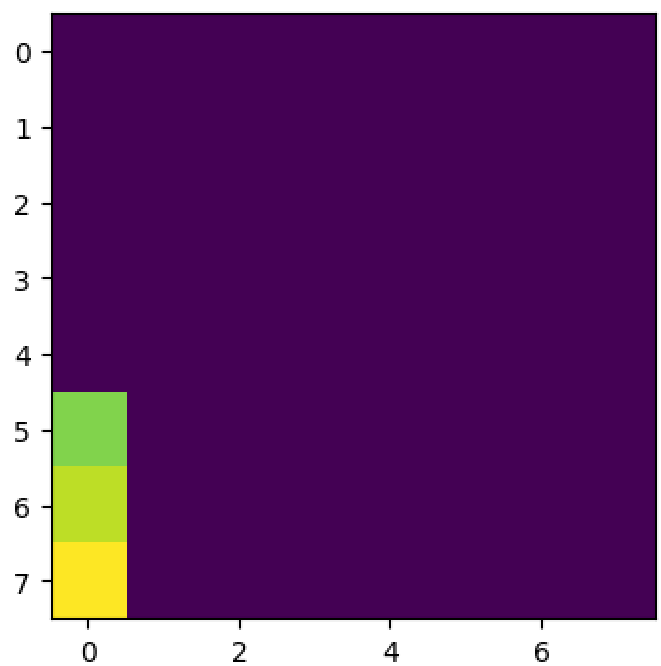
page number 2



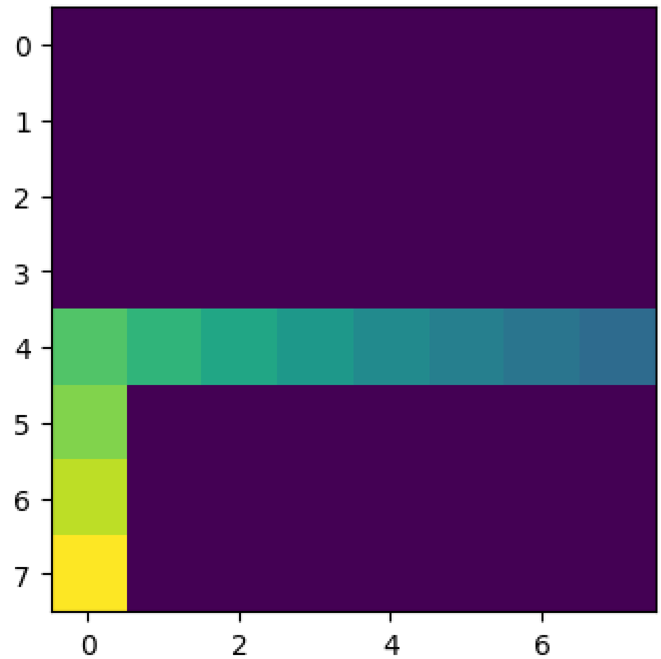
page number 3



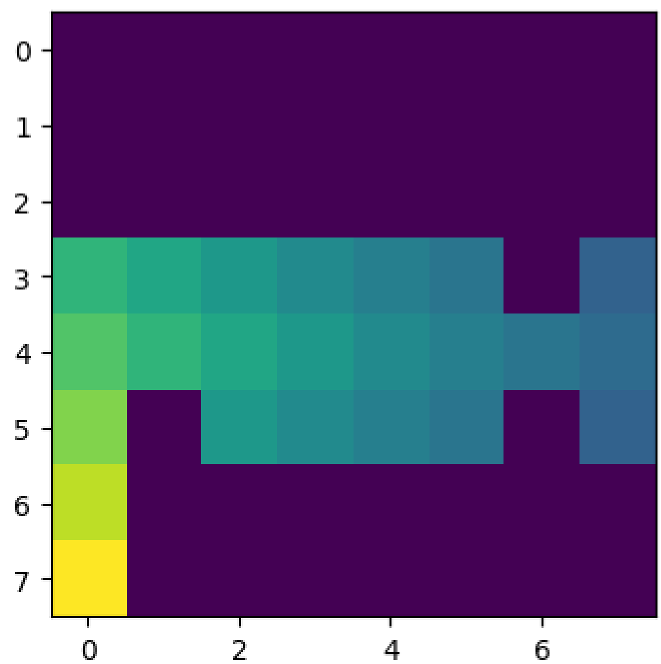

```
page number 4
```



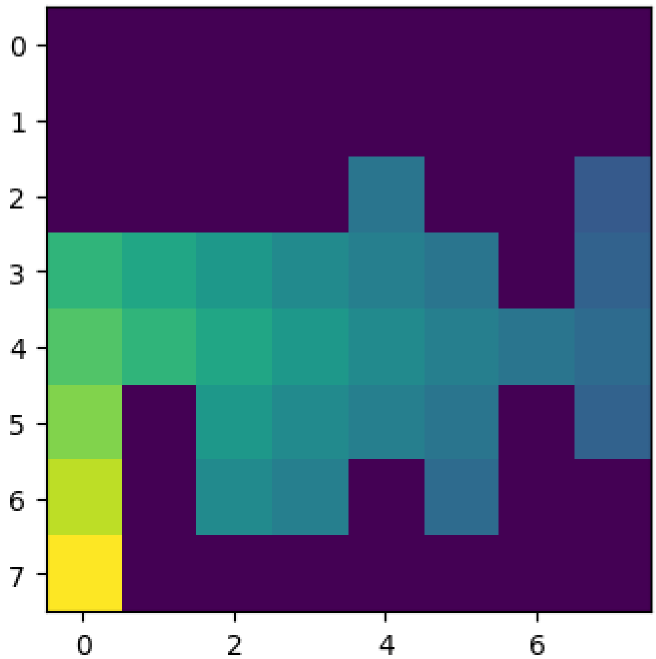

```
page number 5
```



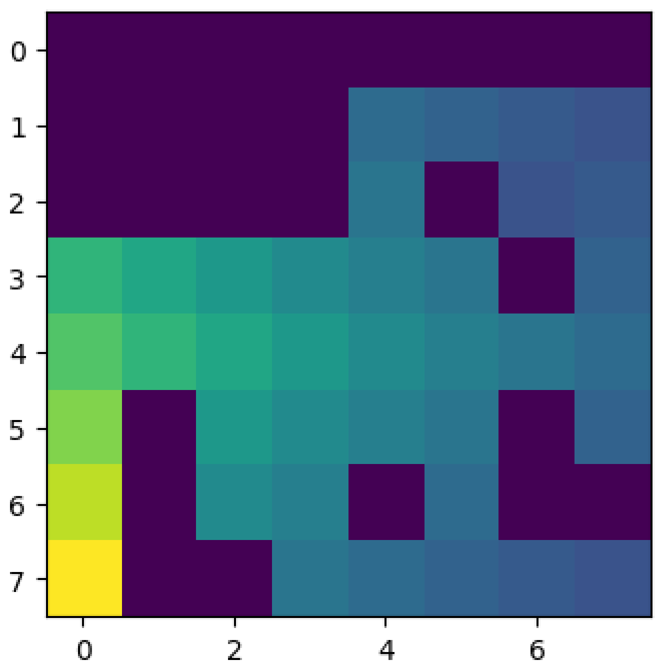
page number 6



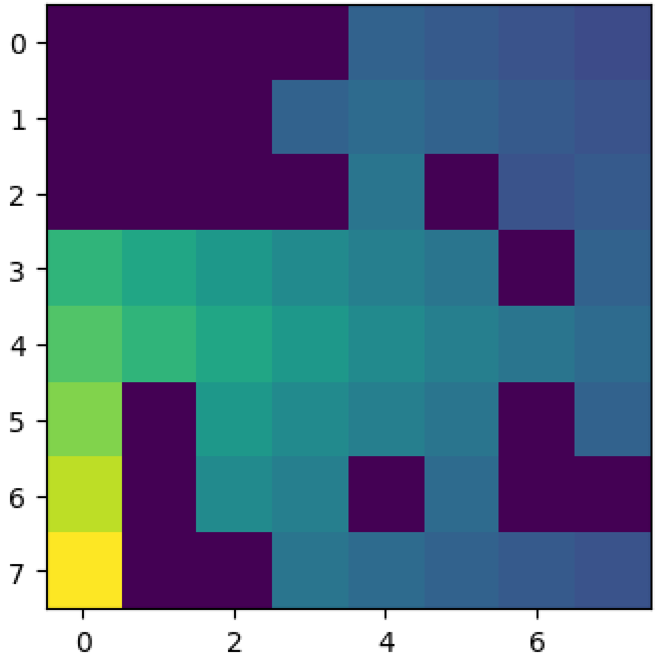
page number 7



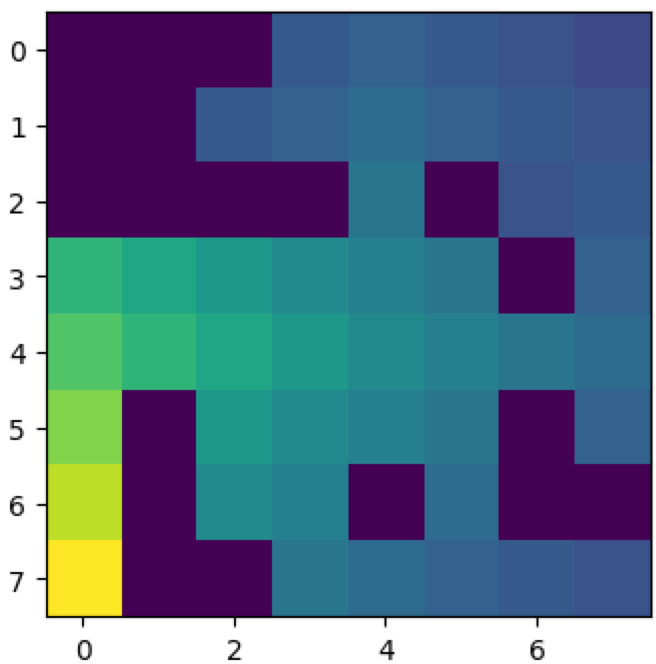

```
page number 8
```



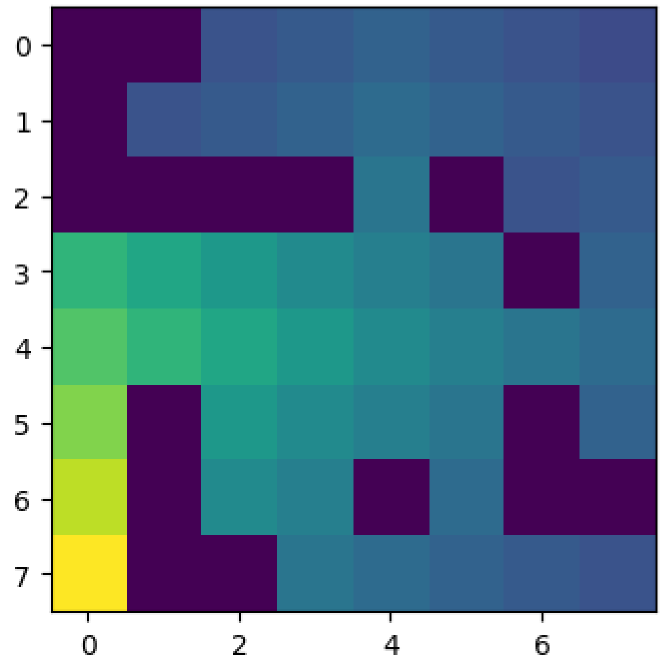

```
page number 9
```



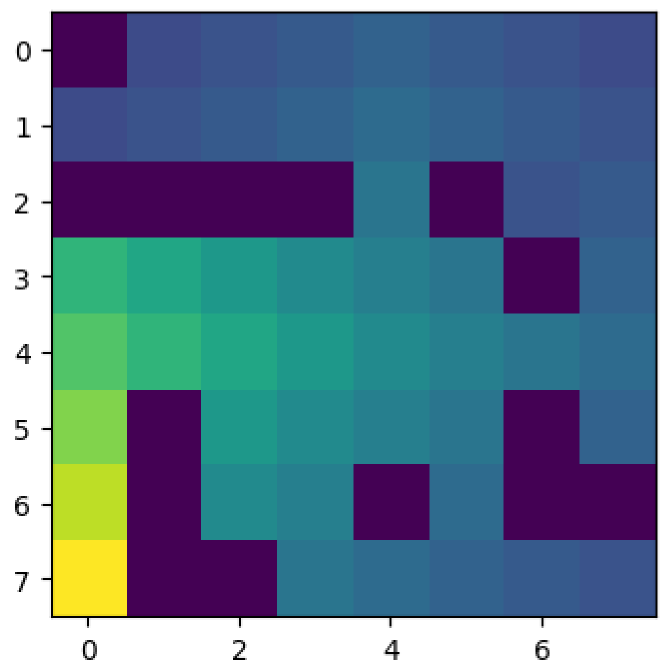
page number 10



page number 11



page number 12



Note that both value and policy iteration can solve the `Frozen Lake` environment (if one can't, you've done something wrong).

Questions *(12 pts)*:

- How many iterations (i.e. policy improvement steps) were required by policy iteration?
  - **Answer:** 12
- How many policy improvement steps were required by value iteration?
  - **Answer:** 17
- If one method took longer to converge, postulate an explanation for this.
  - **Answer**: Because inside of a policy improvement, the correct value function / value table of this policy is computed until convergence, while during value iteration at a specific step the policy and value function do not match

One common benchmark for reinforcment learning algorithms is *sample complexity*: the number of interactions the agent must have with the environment to learn a policy. In policy iteration, we can approximate this as the number of "actions" for which policy improvement is run.

Question: Compute the sample complexity to solve 8x8 FrozenLake. What number do you get? (10 pts)

Answer: It should be the number of policy evaluation steps *number of states inside a step = 145* 8 * 8 = 9280

Q-learning

In the above two algorithms, we had access to the entire MDP: all of the states, with all of the transition probabilities between them. Unfortunately, this is usually not the case.

Below we will implement Q-learning, which is *model free*: it does not require full knowledge of environment dynamics, and instead will try to learn a policy purely through exploration and exploitation.

Fill in the missing functions in the `QLearningAgent` below.

(50 pts)

In [ ]:
```
@dataclass
class QLearningAgent:
    env: gym.Env
    learning_rate: float
    gamma: float
    initial_epsilon: float
    min_epsilon: float
    max_decay_episodes: int
    init_q_value: float = 0.

    def __post_init__(self):
        self.num_states = env.nS
        self.reset()

    def decay_epsilon(self):
        ### TODO: decay epsilon, called after every episode. ################
        if self.epsilon > self.min_epsilon:
          self.epsilon = self.epsilon - self.ep_reduction
        return
        ####################################################################

    def reset(self):
        self.epsilon = self.initial_epsilon
        self.ep_reduction = (self.epsilon - self.min_epsilon) / float(self.max
_decay_episodes)
        self.Q = np.ones((self.num_states, self.env.nA)) * self.init_q_value

    def update_Q(self, state, next_state, action, reward, done):
        ### TODO: update self.Q given new experience. #######################
        if done:
            self.Q[state,action] = reward
        else:
            target = reward + self.gamma*np.max(self.Q[next_state])
            error = target - self.Q[state,action]
            self.Q[state,action] = self.Q[state,action] + self.learning_rate *
error
        ####################################################################

    def get_action(self, state):
        ### TODO: select an action given self.Q and self.epsilon ############
        if np.random.random() < self.epsilon:
          selected_action = np.random.randint(low = 0, high = self.env.nA)
        else:
          selected_action = np.random.choice(np.where(self.Q[state] == self.Q[
state].max())[0])
        return selected_action
        ####################################################################
```

The below code is scaffolding to instantiate and run the above Q-Learning agent. Feel free to examine it to help you implement `QLearningAgent` .

In [ ]:
```
@dataclass
class QLearningEngine:
    env: gym.Env
    agent: Any
    max_episodes: int

    def run(self, n_runs=1):

        rewards = []
        log = []
        for i in tqdm(range(n_runs), desc='Runs'):
            num_actions = 0
            ep_rewards = []
            self.agent.reset()
            # we plot the smoothed return values
            smooth_ep_return = deque(maxlen=100)
            for t in tqdm(range(self.max_episodes), desc='Episode'):
                state = self.env.reset()
                ret = 0
                while True:
                    num_actions += 1
                    action = self.agent.get_action(state)
                    next_state, reward, done, info = self.env.step(action)
                    true_done = done and not info.get('TimeLimit.truncated', F
alse)
                    self.agent.update_Q(state, next_state, action, reward, tru
e_done)
                    ret += reward
                    state = next_state
                    if done:
                        break
                self.agent.decay_epsilon()
                smooth_ep_return.append(ret)
                ep_rewards.append(np.mean(smooth_ep_return))
            rewards.append(ep_rewards)
            run_log = pd.DataFrame({'return': ep_rewards,
                                    'episode': np.arange(len(ep_rewards)),
                                    'iqv': self.agent.init_q_value})
            log.append(run_log)
            print(num_actions)
        return log

def qlearning_sweep(init_q_values, n_runs=4, max_episodes=60000, epsilon=0.8,
learning_rate=0.8):
    logs = dict()
    pbar = tqdm(init_q_values)
    agents = []
    for iqv in pbar:
        pbar.set_description(f'Initial q value:{iqv}')
        env=gym.make('Deterministic-8x8-FrozenLake-v0')
        agent = QLearningAgent(env=env,
                               learning_rate=learning_rate,
                               gamma=0.99,
                               initial_epsilon=epsilon,
                               min_epsilon=0.0,
                               max_decay_episodes=max_episodes,
```

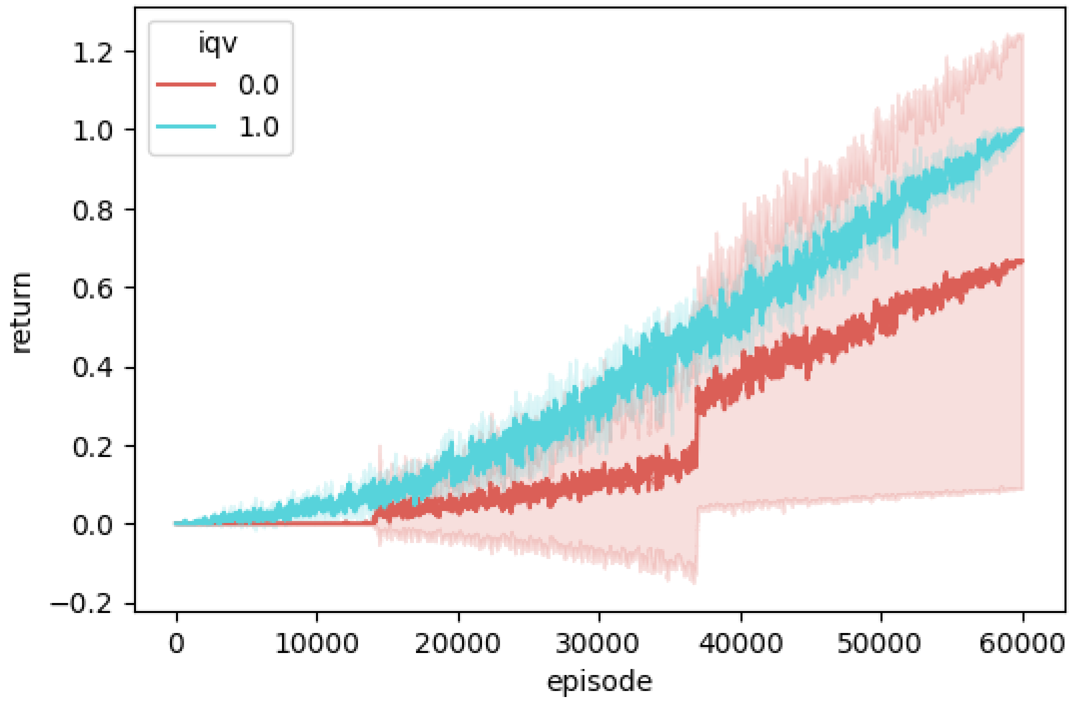
```
                               init_q_value=iqv)
        engine = QLearningEngine(env=env, agent=agent, max_episodes=max_episod
es)
        ep_log = engine.run(n_runs)
        ep_log = pd.concat(ep_log, ignore_index=True)
        logs[f'{iqv}'] = ep_log

        agents.append(agent)
    logs = pd.concat(logs, ignore_index=True)
    return logs, agents
```

Once the agent is implemented, run the below code to try it out on FrozenLake!

In [ ]:
```
init_q_values = [0.,1.] # if it's 0, there is a chance that it can solve the p
roblem.
logs, agents = qlearning_sweep(init_q_values, n_runs=3, max_episodes=60000, ep
silon=0.8)
```

```
842454

835555

825274


804023

801656

801161
```

In [ ]:
```
plot(logs, x_key='episode', y_key='return', legend_key='iqv', estimator='mean'
, ci='sd')
```



Questions

Using the above parameters (`init_q_values = [0., 1.]` , `epsilon=0.8`):

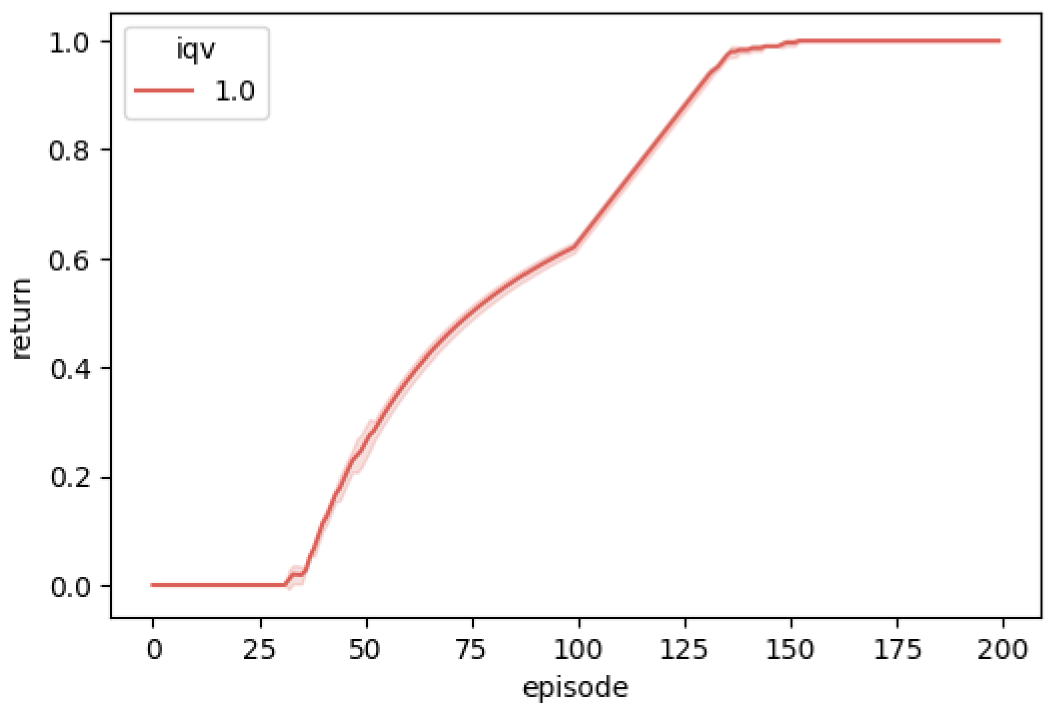
- Print the policy for agents with each of the `init_q_values` . (6 pts)
- For each initial Q value, do you converge to a successful policy (yes/no)? (4 pts)
    - **Answer:** For initial value 1 the agent converges. I am not sure whether agent with initial value 0 it converges or not since sometimes it converges to 1 but other times it is around 0 learning nothing. It is not stable anyway.
- How does the performance compare between the initial Q values? Why is there a difference if any? (5 pts)
    - **Answer:** Initial value 1 converges more stably compared with initial value 0. A potential reason is that with the intial values to be 1, the agent learns to identify "holes" faster than initial values being 0, since initial values are different from the reward got from holes now. Although with initial value 0 the agent should learn to identify the target faster, but since this specific environment has much more holes than targets, initial value 1 learns better.

If you set `epsilon=0.0` and `init_q_value=1` :

- How many steps does the policy take now? Why is it converging faster / slower as compared to higher value of initial_epsilon? (5 pts)
    - **Answer:** The policy now takes only few steps; essentially by setting max_episodes we can see it converges within 100 steps. It is converging much faster because at the "boundary" state of holes it quickly identifies holes and perfectly avoid it. The implementation to ramdonly pick a action when there is a tie actually works as an "exploration" since at the beginning runs every actions for every states have the same value.

In [ ]:
```
### TODO: add policy for agents with each of the init q_values. ########
init_q_values = [1.] # if it's 0, there is a chance that it can solve the prob
lem.
logs, agents = qlearning_sweep(init_q_values, n_runs=3, max_episodes=200, epsi
lon=0.0)
plot(logs, x_key='episode', y_key='return', legend_key='iqv', estimator='mean'
, ci='sd')
########################################################################
```

```
3928

3885

3916
```



Q-Learning Sample Complexity

Remember that we computed the *sample complexity* of Policy Iteration on 8x8 FrozenLake.

Modify the `QLearningEngine` to compute the sample complexity under the same definition (the number of actions the agent must take to learn an optimal policy). For our purposes, we'll define an optimal policy as when the average episode reward is around 1.

Questions:

- What is the (rough) sample complexity of Q-Learning on 8x8 FrozenLake? (10 pts)
    - **Answer**: It is something like 800000+ but it depends on the max_iteration that you set.
- Compare the computed sample complexity of Q-Learning to Policy Iteration. Discuss why these numbers may be so different, or so similar (20 pts).
    - **Answer**: They are different because the Q-Learning has no information about the transition probability and is model-free. Policy Iteration and Value Iteration can access to the transition probabilities of a specific (state, action, next_state) tuple so that it can efficiently scan every state to do updates.

Optional, bonus points

Please enter the bonus code you get after filling out the survey of this assignment. The link to the survey is pinned on Piazza. (10 pts)

Bonus code: volatile-elevator-headroom

End of PSET!