MVA - Homework 1 - Reinforcement Learning (2022/2023)

Name: LAST NAME First Name

Instructions

- The deadline is **November 10 at 11:59 pm (Paris time).**
- By doing this homework you agree to the late day policy, collaboration and misconduct rules reported on [Piazza](https://piazza.com).
- **Mysterious or unsupported answers will not receive full credit**. A correct answer, unsupported by calculations, explanation, or algebraic work will receive no credit; an incorrect answer supported by substantially correct calculations and explanations might still receive partial credit.
- Answers should be provided in **English**.

Colab setup

```
from IPython import get_ipython

if 'google.colab' in str(get_ipython()):
  # install rlberry library
  !pip install git+https://github.com/rlberry-py/rlberry.git@mva2021#egg=rlberry[default]

  # install ffmpeg-python for saving videos
  !pip install ffmpeg-python > /dev/null 2>&1

  # packages required to show video
  !pip install pyvirtualdisplay > /dev/null 2>&1
  !apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1

  print("Libraries installed, please restart the runtime!")
```

```
Libraries installed, please restart the runtime!
```

```
# Create directory for saving videos
!mkdir videos > /dev/null 2>&1

# Initialize display and import function to show videos
import rlberry.colab_utils.display_setup
from rlberry.colab_utils.display_setup import show_video
```

```
# Useful libraries
import numpy as np
import matplotlib.pyplot as plt
```

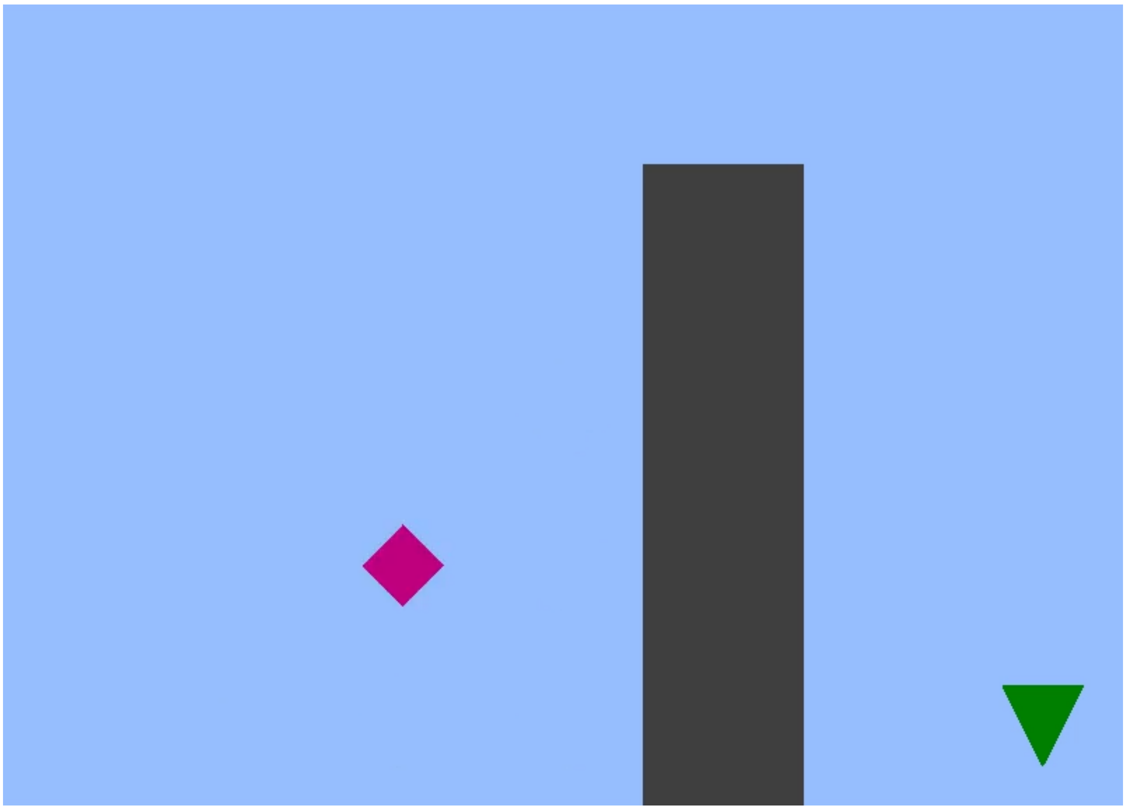
Preparation

In the coding exercises, you will use a *grid-world* MDP, which is represented in Python using the interface provided by the Gym library. The cells below show how to interact with this MDP and how to visualize it.

```
from rlberry.envs import GridWorld

def get_env():
  """Creates an instance of a grid-world MDP."""
  env = GridWorld(
      nrows=5,
      ncols=7,
      reward_at = {(0, 6):1.0},
      walls=((0, 4), (1, 4), (2, 4), (3, 4)),
      success_probability=0.9,
      terminal_states=((0, 6),)
  )
  return env

def render_policy(env, policy=None, horizon=50):
  """Visualize a policy in an environment

  Args:
    env: GridWorld
        environment where to run the policy
    policy: np.array
        matrix mapping states to action (Ns).
        If None, runs random policy.
    horizon: int
        maximum number of timesteps in the environment.
  """
  env.enable_rendering()
  state = env.reset()                       # get initial state
  for timestep in range(horizon):
      if policy is None:
        action = env.action_space.sample()  # take random actions
      else:
        action = policy[state]
      next_state, reward, is_terminal, info = env.step(action)
      state = next_state
      if is_terminal:
        break
  # save video and clear buffer
  env.save_video('./videos/gw.mp4', framerate=5)
  env.clear_render_buffer()
  env.disable_rendering()
  # show video
```

```
show_video('./videos/gw.mp4')
```

```
# Create an environment and visualize it
env = get_env()
render_policy(env)  # visualize random policy

# The reward function and transition probabilities can be accessed through
# the R and P attributes:
print(f"Shape of the reward array = (S, A) = {env.R.shape}")
print(f"Shape of the transition array = (S, A, S) = {env.P.shape}")
print(f"Reward at (s, a) = (1, 0): {env.R[1, 0]}")
print(f"Prob[s\'=2 | s=1, a=0]: {env.P[1, 0, 2]}")
print(f"Number of states and actions: {env.Ns}, {env.Na}")

# The states in the griworld correspond to (row, col) coordinates.
# The environment provides a mapping between (row, col) and the index of
# each state:
print(f"Index of state (1, 0): {env.coord2index[(1, 0)]}")
print(f"Coordinates of state 5: {env.index2coord[5]}")
```



```
Shape of the reward array = (S, A) = (31, 4)
Shape of the transition array = (S, A, S) = (31, 4, 31)
Reward at (s, a) = (1, 0): 0.0
Prob[s'=2 | s=1, a=0]: 0.04999999999999999
Number of states and actions: 31, 4
Index of state (1, 0): 6
Coordinates of state 5: (0, 6)
```

Part 1 - Dynamic Programming

Question 1.1

Consider a general MDP with a discount factor of $\gamma < 1$. Assume that the horizon is infinite (so there is no termination). A policy π in this MDP induces a value function V^π. Suppose an affine transformation is applied to the reward, what is the new value function? Is the optimal policy preserved?

Answer

Question 1.1

We have value function for infinite horizon $V^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s;$ $a_t \sim d_t(h_t), \pi]$. If we perform affine transformation: $r(s_t, a_t) \to a r(s_t, a_t) + b$, we get $V^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t \{a r(s_t, a_t) + b\}] =$
$= E[\sum_{t=0}^{\infty} \gamma^t a r(s_t, a_t)] + E[\sum_{t=0}^{\infty} \gamma^t b] =$
$= a E[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)] + \frac{b}{1-\gamma}$. If $a = -1$, $b = 0$, then the initial optimal policy $\pi^* \in$ $\in \arg\max_{\pi \in \Pi} V^{\pi}$ has to be changed to $\pi^{*-} \in \arg\max_{\pi \in \Pi} -V^{\pi}$ and $\pi^{*-} \in \arg\min_{\pi \in \Pi} V^{\pi}$, ~~obviously~~ π^* and π^{*-} are different.

Signature de l'utilisateur·trice / User's signature

Témoin (prénom, nom) / Witness (first name, last name)

Question 1.2

Consider an infinite-horizon γ-discounted MDP. We denote by Q^* the Q-function of the optimal policy π^*. Prove that, for any function $Q(s,a)$ (which is **not** necessarily the value function of a policy), the following inequality holds for any state s:

$$V^{\pi_Q}(s) \geq V^*(s) - \frac{2}{1-\gamma}||Q^* - Q||_\infty,$$

where $||Q^* - Q||_\infty = \max_{s,a} |Q^*(s,a) - Q(s,a)|$ and $\pi_Q(s) \in \arg\max_a Q(s,a)$. Can you use this result to show that any policy π such that $\pi(s) \in \arg\max_a Q^*(s,a)$ is optimal?

Answer

Question 1.2

Suppose we have s_1, at which $|Q^*(s,\pi^*(s)) - Q(s,\pi_Q(s))|$ has max. value. Therefore, $|Q^*(s_1,\pi^*(s_1)) - Q(s_1,\pi_Q(s_1))| \geq |Q^*(s,\pi(s)) - Q(s,\pi_Q(s))|$, $\forall s \in S$. We have $a_1 = \pi^*(s_1)$, $a_2 = \pi_Q(s_1)$, $\pi_Q(s) = \arg\max_a Q(s,a)$.

$Q(s,a_1) \leq Q(s,a_2)$, $|Q^*(s,a) - Q(s,a)| \leq \|Q^* - Q\|_\infty$ $\forall a \in A$, $\forall s \in S$.

$Q^*(s_1,a_1) - \|Q^* - Q\|_\infty \leq Q(s_1,a_1) \leq$
$\leq Q^*(s_1,a_1) + \|Q^* - Q\|_\infty$

$Q^*(s_1,a_2) - \|Q^* - Q\|_\infty \leq Q^*(s_1,a_2) \leq$
$Q^*(s_1,a_2) + \|Q^* - Q\|_\infty \implies Q^*(s_1,a_1) - \|Q^* - Q\|_\infty \leq Q^*(s_1,a_2) + \|Q^* - Q\|_\infty$.

Let's rewrite Q.

$r(s_1,a_1) + \gamma \sum_y p(y|s_1,a_1) V^*(y) - \|Q^* - Q\|_\infty \leq$
$\leq r(s_1,a_2) + \gamma \sum_y p(y|s_1,a_2) V^*(y) + \|Q^* - Q\|_\infty$

$\implies r(s_1,a_1) - r(s_1,a_2) \leq \gamma \sum_y V^*(y)(p(y|s_1,a_2) -$

Signature de l'utilisateur·trice / User's signature

Date __/__/__

Témoin (prénom, nom) / Witness (first name, last name)

Date __/__/__ Signature

10

$-p(y|S_1,a_1))+2\|Q^*-Q\|_\infty$ (1)

Now, let's look at the difference.

$V^*(S_1)-V^{\pi_Q}(S_1)=r(S_1,a_1)+\gamma\sum_y p(y|S_1,a_1)V^*(y)$ $\underset{\text{use (1)}}{}$

$-r(S_1,a_2)-\gamma\sum_y p(y|S_1,a_2)V^{\pi_Q}(y)\Rightarrow$

$V^*(S_1)-V^{\pi_Q}(S_1)\le\gamma\sum_y V^*(y)(p(y|S_1,a_2)-p(y|S_1,a_1))$

$+2\|Q^*-Q\|_\infty+\gamma\sum_y p(y|S_1,a_1)V^*y-$

$-\gamma\sum_y p(y|S_1,a_2)V^{\pi_Q}(y)=\gamma\sum_y V^*(y)p(y|S_1,a_2)$

$-\gamma\sum_y V^{\pi_Q}(y)p(y|S_1,a_2)+2\|Q^*-Q\|_\infty=$

$=\sum_y p(y|S_1,a_2)(V^*(y)-V^{\pi_Q}(y))+2\|Q^*-Q\|_\infty$

Because S_1 maximizes difference $\Rightarrow$

$V^*(S_1)-V^{\pi_Q}(S_1)\le\sum_y p(y|S_1,a_2)(V^*(y)-V^{\pi_Q}(y))+$

$+2\|Q^*-Q\|_\infty\le\gamma(V^*(S_1)-V^{\pi_Q}(S_1))\underbrace{\sum p(y|S_1,a_2)}_{1}+$

$2\|Q^*-Q\|_\infty\Rightarrow V^*(S_1)-V^{\pi_Q}(S_1)\le$

$\le(V^*(S_1)-V^{\pi_Q}(S_1))\gamma+2\|Q^*-Q\|_\infty\Rightarrow$

$(1-\gamma)(V^*(S_1)-V^{\pi_Q}(S_1))\le2\|Q^*-Q_\infty\|$

$V^*(S_1)-V^{\pi_Q}(S_1)\le\frac{2}{1-\gamma}\|Q^*-Q_\infty\|$, S_1 maximizes difference $\Rightarrow V^*(S)-V^{\pi_Q}(S)\le$

$\le V^*(S_1)-V^{\pi_Q}(S_1)\le\frac{2}{1-\gamma}\|Q^*-Q_\infty\|\Rightarrow$

Signature de l'utilisateur·trice / User's signature

Date __/__/__

Témoin (prénom, nom) / Witness (first name, last name)

Date __/__/__ Signature 11

$\Rightarrow V^{\pi_Q}(s) \geq V^*(s) - \frac{2}{1-\gamma} \|Q^* - Q\|_\infty$

If $\pi(s) \in \arg\max_a Q^*(s,a)$ and

$\pi(s) \in \arg\max_a Q(s,a) \Rightarrow$

$\|Q^* - Q\|_\infty = 0 \Rightarrow V^{\pi_Q}(s) \geq V^*(s) \Rightarrow$

$\Rightarrow \pi(s)$ - optimal policy

Question 1.3

In this question, you will implement and compare the policy and value iteration algorithms for a finite MDP.

Complete the functions `policy_evaluation`, `policy_iteration` and `value_iteration` below.

Compare value iteration and policy iteration. Highlight pros and cons of each method.

Answer

In policy iteration and values interation we are looping through different things. At policy iteration, we using policy evaluation step in order to get bigger value function. For value iteration, we looping over optimal value function. Policy iteration is faster than value iteration, as a policy iteration converges more quickly than a value function. For large (but still discrete) MDPs, we need to select the number of iterations for policy evaluation, which is a hyperparameter in policy iteration. In contrast, value iteration combines the evaluation and improvement in one step by using the Bellman optimality equation, which removes the need for this hyperparameter.

```
def policy_evaluation(P, R, policy, gamma=0.9, tol=1e-2):
    """
    Args:
        P: np.array
            transition matrix (NsxNaxNs)
        R: np.array
            reward matrix (NsxNa)
        policy: np.array
            matrix mapping states to action (Ns)
        gamma: float
            discount factor
        tol: float
            precision of the solution
    Return:
        value_function: np.array
            The value function of the given policy
    """
    Ns, Na = R.shape
    # ====================================================
      # YOUR IMPLEMENTATION HERE
    #
    value_function = np.zeros(Ns)
    R1 = np.zeros(Ns)
    P1 = np.zeros((Ns,Ns))
    for s in range(Ns):
        a = policy[s]
        R1[s] = R[s,a]
        for ns in range(Ns):
            P1[s,ns] = P[s,a,ns]
    value_function = np.linalg.solve(np.eye(Ns)- gamma*P1, R1)
    # ====================================================
    return value_function
```

```
from re import I
def policy_iteration(P, R, gamma=0.9, tol=1e-3):
    """
    Args:
        P: np.array
            transition matrix (NsxNaxNs)
        R: np.array
            reward matrix (NsxNa)
        gamma: float
            discount factor
        tol: float
            precision of the solution
    Return:
        policy: np.array
            the final policy
        V: np.array
            the value function associated to the final policy
    """
    Ns, Na = R.shape
    V = np.zeros(Ns)
    policy = np.ones(Ns, dtype=int)
    # ====================================================
      # YOUR IMPLEMENTATION HERE
    #
    policy1 = policy.copy()
    V1 = policy_evaluation(P, R, policy1)
    V = policy_evaluation(P, R, policy)
    i = 0
    while i == 0 or np.linalg.norm(V - V1, np.inf)>tol:
        policy1 = policy.copy()
        V1= V.copy()
        i += 1
        for s in range(Ns):
            policy[s] = np.argmax(R[s,:] + gamma*P[s,:,:].dot(V))
        V = policy_evaluation(P, R, policy, gamma)
    # ====================================================
    return policy, V



def value_iteration(P, R, gamma=0.9, tol=1e-3):
    """
    Args:
        P: np.array
            transition matrix (NsxNaxNs)
        R: np.array
            reward matrix (NsxNa)
        gamma: float
            discount factor
        tol: float
            precision of the solution
    Return:
        Q: final Q-function (at iteration n)
        greedy_policy: greedy policy wrt Qn
```

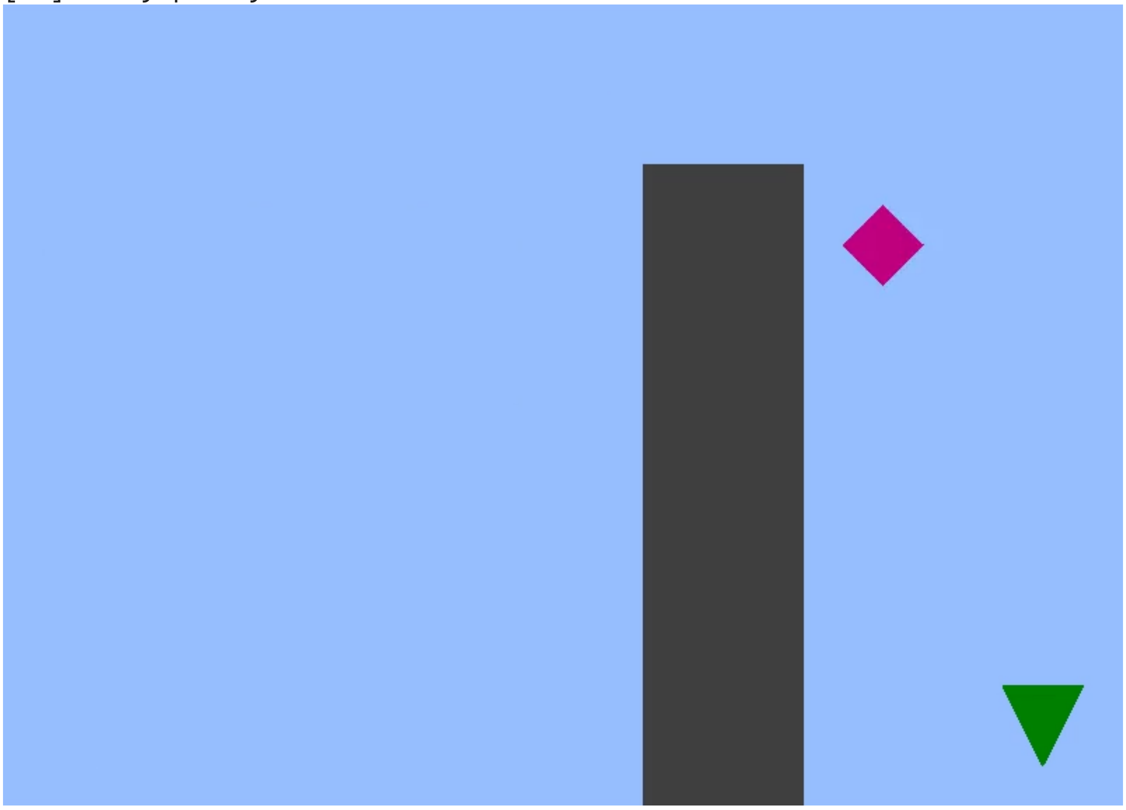
```
        Qfs: all Q-functions generated by the algorithm (for visualization)
    """
    Ns, Na = R.shape
    Q = np.zeros((Ns, Na))
    Qfs = [Q]
    # ====================================================
      # YOUR IMPLEMENTATION HERE
    #
    greedy_policy = np.argmax(Q, axis=1)
    V1 = np.zeros(Ns)
    V = np.zeros(Ns)
    i = 0
    while i == 0 or np.linalg.norm(V - V1, np.inf)>tol:
        V1 = V.copy()
        i +=1
        for s in range(Ns):
            for a in range(Na):
                Q[s,a] = R[s,a]+ gamma*(P[s,a,:].dot(V))
            V[s] = np.max(Q[s,:])
        Qfs.append(Q.copy())
    greedy_policy = np.argmax(Q,axis = 1)
    Qfs[0] = np.zeros((Ns, Na))
    # ====================================================
    return Q, greedy_policy, Qfs
```

Testing your code

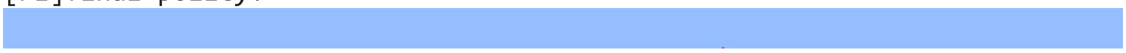
```
# Parameters
tol = 1e-5
gamma = 0.99

# Environment
env = get_env()

# run value iteration to obtain Q-values
VI_Q, VI_greedypol, all_qfunctions = value_iteration(env.P, env.R, gamma=gamma, tol=tol)

# render the policy
print("[VI]Greedy policy: ")
render_policy(env, VI_greedypol)

# compute the value function of the greedy policy using matrix inversion
# ====================================================
# YOUR IMPLEMENTATION HERE
# compute value function of the greedy policy
#
greedy_V = policy_evaluation(env.P,env.R, VI_greedypol, gamma=gamma, tol= tol )
# ====================================================

# show the error between the computed V-functions and the final V-function
# (that should be the optimal one, if correctly implemented)
# as a function of time
final_V = all_qfunctions[-1].max(axis=1)
```

```
norms = [ np.linalg.norm(q.max(axis=1) - final_V) for q in all_qfunctions]
plt.plot(norms)
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.title("Value iteration: convergence")

#### POLICY ITERATION ####
PI_policy, PI_V = policy_iteration(env.P, env.R, gamma=gamma, tol=tol)
print("\n[PI]final policy: ")
render_policy(env, PI_policy)

## Uncomment below to check that everything is correct
assert np.allclose(PI_policy, VI_greedypol)#,\
#     "You should check the code, the greedy policy computed by VI is not equal to the sol
assert np.allclose(PI_V, greedy_V)#,\
#     "Since the policies are equal, even the value function should be"

plt.show()
```

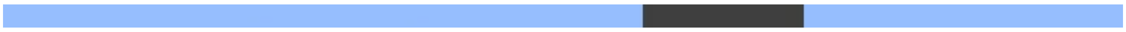
```
[VI]Greedy policy:
```



```
[PI]final policy:
```



Part 2 - Tabular RL



Question 2.1

The code below collects two datasets of transitions (containing states, actions, rewards and next states) for a discrete MDP.

For each of the datasets:

1. Estimate the transitions and rewards, $\hat{P}$ and $\hat{R}$.
2. Compute the optimal value function and the optimal policy with respect to the estimated MDP (defined by $\hat{P}$ and $\hat{R}$), which we denote by $\hat{\pi}$ and $\hat{V}$.
3. Numerically compare the performance of $\hat{\pi}$ and $\pi^\star$ (the true optimal policy), and the error between $\hat{V}$ and V^* (the true optimal value function).

Which of the two data collection methods do you think is better? Why?

Answer

[answer last question + implementation below] The second method produces better results and in terms of policy comparison and in terms of error between value functions. I think second method is better, because during the random policy method we can just miss and get zero reward, especially at low number of samples. So the second method is more reliable. This conclusion is supported by the results.

Iteration

```
def get_random_policy_dataset(env, n_samples):
  """Get a dataset following a random policy to collect data."""
  states = []
  actions = []
  rewards = []
  next_states = []

  state = env.reset()
  for _ in range(n_samples):
    action = env.action_space.sample()
    next_state, reward, is_terminal, info = env.step(action)
    states.append(state)
    actions.append(action)
    rewards.append(reward)
    next_states.append(next_state)
    # update state
    state = next_state
    if is_terminal:
      state = env.reset()

  dataset = (states, actions, rewards, next_states)
  return dataset

def get_uniform_dataset(env, n_samples):
  """Get a dataset by uniformly sampling states and actions."""
  states = []
  actions = []
  rewards = []
  next_states = []
  for _ in range(n_samples):
    state = env.observation_space.sample()
    action = env.action_space.sample()
    next_state, reward, is_terminal, info = env.sample(state, action)
    states.append(state)
    actions.append(action)
    rewards.append(reward)
    next_states.append(next_state)

  dataset = (states, actions, rewards, next_states)
  return dataset


# Collect two different datasets
num_samples = 500
env = get_env()
dataset_1 = get_random_policy_dataset(env, num_samples)
dataset_2 = get_uniform_dataset(env, num_samples)


# Item 3: Estimate the MDP with the two datasets; compare the optimal value
```

```
# functions in the true and in the estimated MDPs
R1 = np.zeros((31, 4))
P1 = np.zeros((31, 4, 31))
R2 = np.zeros((31, 4))
P2 = np.zeros((31, 4, 31))
for s in range(31):
  for a in range(4):
    sum = 0
    n = 0
    for index in range(num_samples):
      if dataset_1[0][index]==s and dataset_1[1][index]==a:
        sum += dataset_1[2][index]
        n+=1
    if n != 0:
      R1[s, a] = sum/n
for s in range(31):
  for a in range(4):
    sum = 0
    n = 0
    for index in range(num_samples):
      if dataset_2[0][index]==s and dataset_2[1][index]==a:
        sum += dataset_2[2][index]
        n+=1
    if n != 0:
      R2[s, a] = sum/n
for s in range(31):
  for a in range(4):
    for ns in range(31):
      sum = 0
      n = 0
      for index in range(num_samples):
        if dataset_1[0][index]==s and dataset_1[1][index]==a:
          n+=1
        if dataset_1[0][index]==s and dataset_1[1][index]==a and dataset_1[3][index]==ns:
          sum += 1
      if n != 0:
        P1[s, a, ns] = sum/n
for s in range(31):
  for a in range(4):
    for ns in range(31):
      sum = 0
      n = 0
      for index in range(num_samples):
        if dataset_2[0][index]==s and dataset_2[1][index]==a:
          n+=1
        if dataset_2[0][index]==s and dataset_2[1][index]==a and dataset_2[3][index]==ns:
          sum += 1
      if n != 0:
        P2[s, a, ns] = sum/n

PI_policy1, PI_V1 = policy_iteration(P1, R1, gamma=gamma, tol=tol)
PI_policy2, PI_V2 = policy_iteration(P2, R2, gamma=gamma, tol=tol)
print("Optimal policy and value function:")
print(PI_policy, PI_V)
print("Optimal policy and value function dataset 1:")
```

```
print(PI_policy1, PI_V1)
print("Optimal policy and value function dataset 2:")
print(PI_policy2, PI_V2)
print("Policy comparison with dataset 1 and accuracy:")
print(PI_policy1==PI_policy, (PI_policy1==PI_policy).mean())
print("Policy comparison with dataset 2 and accuracy:")
print(PI_policy2==PI_policy, (PI_policy2==PI_policy).mean())
print("Value function error with dataset 1: ", max(PI_V - PI_V1))
print("Value function error with dataset 2: ", max(PI_V - PI_V2))
# ...
```

```
Optimal policy and value function:
[3 1 3 3 1 0 3 3 3 3 2 2 3 3 3 3 2 2 1 1 3 3 2 2 1 1 1 1 1 2 2] [ 85.0635286   85.888
 100.          85.92657801  86.83074166  87.82948871  88.71311516
  97.67341912  98.76349605  86.88867748  87.83667295  88.85592657
  89.81741877  96.5828574   97.54872335  87.8646426   88.85711171
  89.89334278  90.94023704  95.49856884  96.35549348  88.72659644
  89.81817825  90.94027463  92.08143024  93.24168901  94.41709026
  95.20003662]
Optimal policy and value function dataset 1:
[1 1 1 3 0 3 1 3 1 3 3 2 3 1 1 3 1 2 1 3 1 3 2 2 1 1 1 1 1 1 2] [ 85.86299059  86.753
 100.          86.63821934  87.51335287  88.51475571  89.40884415
  95.11322442  99.           87.65395074  88.80832965  89.70538348
  90.61149847  96.07396406  98.01         88.53934418  89.4466192
  90.61149847  91.52676613  95.24822917  96.48306938  89.38192814
  90.28477589  91.41550884  92.68125148  93.61742573  94.5630563
  95.51823868]
Optimal policy and value function dataset 2:
[0 0 0 0 1 0 0 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] [ -0.    -0.    -0.
  98.01  99.    -0.    -0.    -0.    -0.    -0.     0.    -0.    -0.
  -0.    -0.    -0.     0.    -0.     0.     0.    -0.    -0.    -0.
   0.  ]
Policy comparison with dataset 1 and accuracy:
[False  True False  True False False False  True False  True False  True
  True False False  True False  True  True False False  True  True  True
  True  True  True  True  True False  True] 0.5806451612903226
Policy comparison with dataset 2 and accuracy:
[False False False False  True  True False False False False False  True
 False False False False False False False False False False False False
 False False False False False False False] 0.0967741935483871
Value function error with dataset 1:  98.76966849322514
Value function error with dataset 2:  97.54872335128597
```

Question 2.2

Suppose that $\hat{P}$ and $\hat{R}$ are estimated from a dataset of exactly N i.i.d. samples from **each** state-action pair. This means that, for each (s, a), we have N samples $\{(s'_1, r_1, \ldots, s'_N, r_N\}$, where $s'_i \sim P(\cdot|s, a)$ and $r_i \sim R(s, a)$ for $i = 1, \ldots, N$, and

$$\hat{P}(s'|s,a) = \frac{1}{N}\sum_{i=1}^{N} 1(s'_i = s'),$$

$$\hat{R}(s,a) = \frac{1}{N}\sum_{i=1}^{N} r_i.$$

Suppose that R is a distribution with support in $[0, 1]$. Let $\hat{V}$ be the optimal value function computed in the empirical MDP (i.e., the one with transitions $\hat{P}$ and rewards $\hat{R}$). For any $\delta \in (0, 1)$, derive an upper bound to the error

$$\|\hat{V} - V^*\|_\infty$$

which holds with probability at least $1 - \delta$.

Note Your bound should only depend on deterministic quantities like N, γ, δ, S, A. It should *not* dependent on the actual random samples.

Hint The following two inequalities may be helpful.

1. **A (simplified) lemma**. For any state $\bar{s}$,

$$|\hat{V}(\bar{s}) - V^*(\bar{s})| \leq \frac{1}{1-\gamma} \max_{s,a} \left| R(s,a) - \hat{R}(s,a) + \gamma \sum_{s'} (P(s'|s,a) - \hat{P}(s'|s,a)) V^*(s'$$

2. **Hoeffding's inequality**. Let $X_1, \ldots X_N$ be N i.i.d. random variables bounded in the interval $[0, b]$ for some $b > 0$. Let $\bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$ be the empirical mean. Then, for any $\epsilon > 0$,

$$\mathbb{P}(|\bar{X} - \mathbb{E}[\bar{X}]| > \epsilon) \leq 2e^{-\frac{2N\epsilon^2}{b^2}}.$$

Answer

[your derivation here]

Question 2.3

Suppose once again that we are given a dataset of N samples in the form of tuples (s_i, a_i, s_i', r_i). We know that each tuple contains a valid transition from the true MDP, i.e., $s_i' \sim P(\cdot|s_i, a_i)$ and $r_i \sim R(s_i, a_i)$, while the state-action pairs (s_i, a_i) from which the transition started can be arbitrary.

Suppose we want to apply Q-learning to this MDP. Can you think of a way to leverage this offline data to improve the sample-efficiency of the algorithm? What if we were using SARSA instead?

Answer

Q-learning learns action values relative to the greedy policy. One the disadvantages of Q-Learing is that it is not guaranteed to converge when combined with linear approximation. Therefore, we need to fight these problems. In that case let's compare it with SARSA. SARSA learns action values relative to the policy it follows. Both of the algorithms converge to the real value function,

Part 3 - RL with Function Approximation

Question 3.1

Given a datset (s_i, a_i, r_i, s'_i) of (states, actions, rewards, next states), the Fitted Q-Iteration (FQI) algorithm proceeds as follows:

- We start from a Q function $Q_0 \in \mathcal{F}$, where $\mathcal{F}$ is a function space;
- At every iteration k, we compute Q_{k+1} as:

$$Q_{k+1} \in \arg\min_{f \in \mathcal{F}} \frac{1}{2} \sum_{i=1}^{N} \left(f(s_i, a_i) - y_i^k \right)^2 + \lambda \Omega(f)$$

where $y_i^k = r_i + \gamma \max_{a'} Q_k(s'_i, a')$, $\Omega(f)$ is a regularization term and $\lambda > 0$ is the regularization coefficient.

Consider FQI with *linear* function approximation. That is, for a given feature map $\phi : S \rightarrow \mathbb{R}^d$, we consider a parametric family of Q functions $Q_\theta(s, a) = \phi(s)^T \theta_a$ for $\theta_a \in \mathbb{R}^d$. Suppose we are applying FQI on a given dataset of N tuples of the form (s_i, a_i, r_i, s'_i) and we are at the k-th iteration. Let $\theta_k \in \mathbb{R}^{d \times A}$ be our current parameter. Derive the *closed-form* update to find θ_{k+1}, using $\frac{1}{2} \sum_a ||\theta_a||_2^2$ as regularization.

Answer

[your derivation here]

Question 3.2

The code below creates a larger gridworld (with more states than the one used in the previous questions), and defines a feature map. Implement linear FQI to this environment (in the function `linear_fqi()` below), and compare the approximated Q function to the optimal Q function computed with value iteration.

Can you improve the feature map in order to reduce the approximation error?

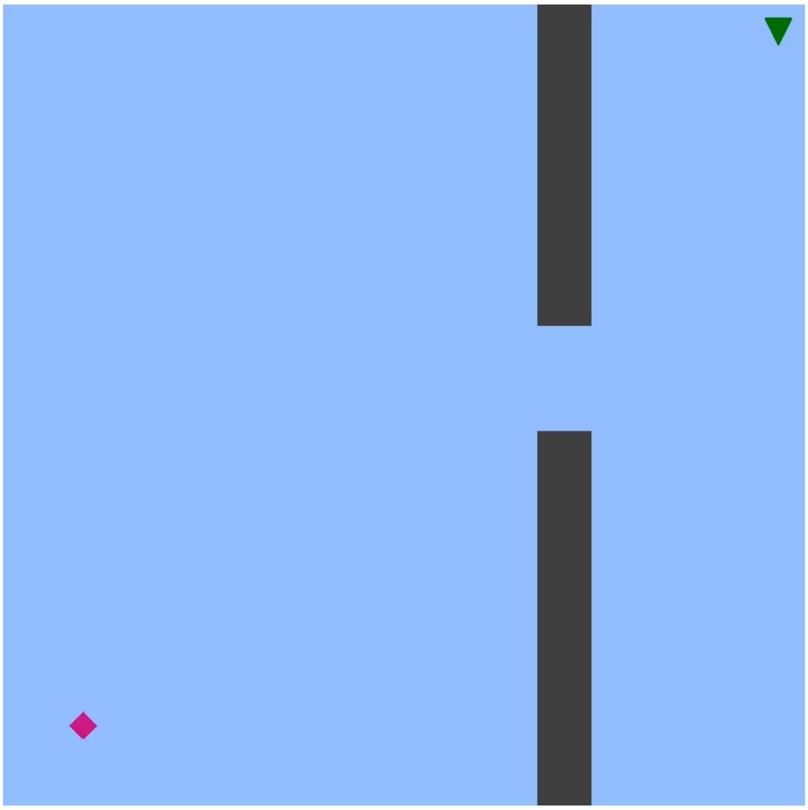
Answer

[explanation about how you tried to reduce the approximation error + FQI implementation below]

```
def get_large_gridworld():
  """Creates an instance of a grid-world MDP with more states."""
```

```
    walls = [(ii, 10) for ii in range(15) if (ii != 7 and ii != 8)]
    env = GridWorld(
        nrows=15,
        ncols=15,
        reward_at = {(14, 14):1.0},
        walls=tuple(walls),
        success_probability=0.9,
        terminal_states=((14, 14),)
    )
    return env


  class GridWorldFeatureMap:
    """Create features for state-action pairs

    Args:
      dim: int
        Feature dimension
      sigma: float
        RBF kernel bandwidth
    """
    def __init__(self, env, dim=15, sigma=0.25):
      self.index2coord = env.index2coord
      self.n_states = env.Ns
      self.n_actions = env.Na
      self.dim = dim
      self.sigma = sigma

      n_rows = env.nrows
      n_cols = env.ncols

      # build similarity matrix
      sim_matrix = np.zeros((self.n_states, self.n_states))
      for ii in range(self.n_states):
          row_ii, col_ii = self.index2coord[ii]
          x_ii = row_ii / n_rows
          y_ii = col_ii / n_cols
          for jj in range(self.n_states):
              row_jj, col_jj = self.index2coord[jj]
              x_jj = row_jj / n_rows
              y_jj = col_jj / n_cols
              dist = np.sqrt((x_jj - x_ii) ** 2.0 + (y_jj - y_ii) ** 2.0)
              sim_matrix[ii, jj] = np.exp(-(dist / sigma) ** 2.0)

      # factorize similarity matrix to obtain features
      uu, ss, vh = np.linalg.svd(sim_matrix, hermitian=True)
      self.feats = vh[:dim, :]

    def map(self, observation):
      feat = self.feats[:, observation].copy()
      return feat

  env = get_large_gridworld()
  feat_map = GridWorldFeatureMap(env)
```

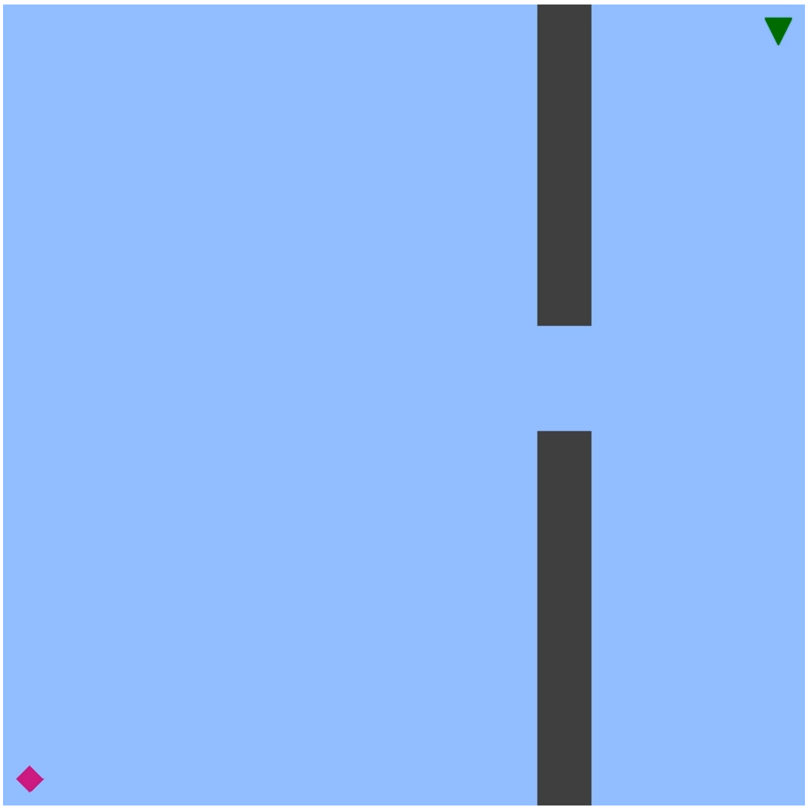
```
# Visualize large gridworld
render_policy(env)

# The features have dimension (feature_dim).
feature_example = feat_map.map(1) # feature representation of s=1
print(feature_example)

# Initial vector theta representing the Q function
theta = np.zeros((feat_map.dim, env.action_space.n))
print(theta.shape)
print(feature_example @ theta) # approximation of Q(s=1, a)
```

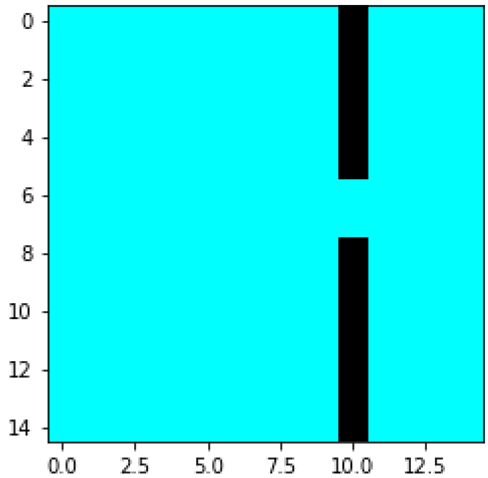


```
[-0.02850699  0.063555   -0.02169407 -0.06441918  0.04505794 -0.07537777
  0.08506473 -0.09325287  0.09644275 -0.00535101  0.11632395 -0.13074085
  0.00921342 -0.13853662  0.07118419]
(15, 4)
[0. 0. 0. 0.]
```

```
def linear_fqi(env, feat_map, num_iterations, lambd=0.1, gamma=0.95):
  """
  # Linear FQI implementation
  # TO BE COMPLETED
  """

  # get a dataset
  #    dataset = get_uniform_dataset(env, n_samples=...)
  # OR dataset = get_random_policy_dataset(env, n_samples=...)


  theta = np.zeros((feat_map.dim, env.Na))

  for it in range(num_iterations):
    # ...
```

```
      pass

    return theta

  # ----------------------------
  # Environment and feature map
  # ----------------------------
  env = get_large_gridworld()
  # you can change the parameters of the feature map, and even try other maps!
  feat_map = GridWorldFeatureMap(env, dim=15, sigma=0.25)

  # -------
  # Run FQI
  # -------
  theta = linear_fqi(env, feat_map, num_iterations=100)

  # Compute and run greedy policy
  Q_fqi = np.zeros((env.Ns, env.Na))
  for ss in range(env.Ns):
    state_feat = feat_map.map(ss)
    Q_fqi[ss, :] = state_feat @ theta

  V_fqi = Q_fqi.max(axis=1)
  policy_fqi = Q_fqi.argmax(axis=1)
  render_policy(env, policy_fqi, horizon=100)

  # Visualize the approximate value function in the gridworld.
  img = env.get_layout_img(V_fqi)
  plt.imshow(img)
  plt.show()
```





✓ 18s completed at 11:05 PM