assignment1

October 22, 2022

1 Assignment 1: Tabular Reinforcement Learning

2022-2023 fall quarter, CS269 Seminar 5: Reinforcement Learning. Department of Computer Science at University of California, Los Angeles. Course Instructor: Professor Bolei ZHOU. Assignment author: Zhenghao PENG.

Student Name	Student ID
Linqiao Jiang	905542286

Welcome to the assignment 1 of our reinforcement learning course. The objective of this assignment is for you to understand the classic methods used in tabular RL.

This assignment has the following sections:

- Section 1: Warm-up on the RL environment (35 points)
- Section 2: Implementation of the model-based family of algorithms: policy iteration and value iteration. (65 points)

You need to go through this self-contained notebook, where **21 TODOs** are scattered in cells with special `[TODO]` signs. You need to finish all TODOs.

You are encouraged to add more code on extra cells at the end of the each section to investigate the problems you think interesting. At the end of the file, we leave a place for you to write comments optionaly (Yes, please give us either negative or positive rewards so we can keep improving the assignment!).

Please report any code bugs to us via **github issues**.

Before you get start, remember to follow the instruction at https://github.com/ucla-rlcourse/assignment-2022fall/tree/main/assignment0 to setup your python environment.

1.1 Dependencies

This assignment requires the following dependencies:

1. `gym<0.20.0`
2. `numpy`
3. `scipy`

You can install all of them through (or simply run next cell):

`pip install 'gym<0.20.0' numpy scipy`

After installation, remember to restart the jupyter notebook kernel by clicking `kernel/restart` at top bar and rerun the cells.

```
[ ]: # You don't have to run this in your local machine
     # if you already installed everything.
     # Please run this cell when using Colab

     !pip install 'gym<0.20.0' numpy scipy
```

```
Requirement already satisfied: gym<0.20.0 in
/Users/linqiaojiang/opt/anaconda3/envs/RL/lib/python3.9/site-packages (0.19.0)
Requirement already satisfied: numpy in
/Users/linqiaojiang/opt/anaconda3/envs/RL/lib/python3.9/site-packages (1.23.3)
Requirement already satisfied: scipy in
/Users/linqiaojiang/opt/anaconda3/envs/RL/lib/python3.9/site-packages (1.9.2)
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in
/Users/linqiaojiang/opt/anaconda3/envs/RL/lib/python3.9/site-packages (from
gym<0.20.0) (1.6.0)
```

Now start running the cells sequentially (by `ctrl + enter` or `shift + enter`) to avoid unnecessary errors by skipping some cells.

1.2 Section 1: Warm-up on the RL environment

(35/100 points)

In this section, we will go through the basic concepts of RL environments using OpenAI Gym. Besides, you will get the first sense of the toy environment we will use in the rest of the assignment.

Every Gym environment should contain the following attributes:

1. `env.step(action)` To advance the environment by one time step through applying `action`. Will return four things: `observation, reward, done, info`, wherein `done` is a boolean value indicating whether this **episode** is finished. `info` is a dict containing some information the user is interested in.
2. `env.reset()` To reset the environment, back to the initial state. Will return the initial observation of the new episode.
3. `env.render()` To render the current state of the environment for human-being
4. `env.action_space` The allowed action format. In our case, it is `Discrete(4)` which means the action is an integer in the range [0, 1, 2, 3]. Therefore the `action` for `step(action)` should obey the limit of the action space.
5. `env.observation_space` The observation space.

Note that the word **episode** means the process that an agent interacts with the environment from the initial state to the terminal state. Within one episode, the agent will only receive one `done=True`, when it goes to the terminal state (the agent is dead or the game is over).

We will use `FrozenLake8x8-v1` as our environment. In this environment, the agent controls the movement of a *character* in a grid world. Some tiles of the grid are walkable, and others are not, making 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. The meaning of each character:

1. S : starting point, safe
2. F : frozen surface, safe
3. H : hole, fall to your doom
4. G : goal, where the frisbee is located

[ ]:
```python
# Run this cell without modification

# Import some packages that we need to use
import gym
import numpy as np
from collections import deque
import time

# Prepare some useful functions
from IPython.display import clear_output
from gym.envs.registration import register

def wait(sleep=0.2):
    clear_output(wait=True)
    time.sleep(sleep)

def print_table(data):
    if data.ndim == 2:
        for i in range(data.shape[1]):
            print("\n=== The state value for action {} ===".format(i))
            print_table(data[:, i])
        return
    assert data.ndim == 1, data
    if data.shape[0] == 16:  # FrozenLake-v0
        text = "+-----+-----+-----+-----+-----+\n" \
               "|     |   0 |   1 |   2 |   3 |\n" \
               "|-----+-----+-----+-----+-----+\n"
        for row in range(4):
            tmp = "| {}   |{:.3f}|{:.3f}|{:.3f}|{:.3f}|\n" \
                  "|     |     |     |     |     |\n" \
                  "+-----+-----+-----+-----+-----+\n" \
                  "".format(
                row, *[data[row * 4 + col] for col in range(4)]
            )
            text = text + tmp
    else:
        text = "+-----+-----+-----State Value Mapping-----+-----+-----+\n" \
               "|     |   0 |   1 |   2 |   3 |   4 |   5 |   6 |   7 |\n" \
```

```
            "|-----+-----+-----+-----+-----+-----+-----+-----+-----|\n"
    for row in range(8):
        tmp = "| {}   |{:.3f}|{:.3f}|{:.3f}|{:.3f}|{:.3f}|{:.3f}|{:.3f}|{" \
              ":.3f}|\n" \
              "|     |     |     |     |     |     |     |     |     |\n" \
              "+-----+-----+-----+-----+-----+-----+-----+-----+-----+\n" \
              "".format(
            row, *[data[row * 8 + col] for col in range(8)]
        )
        text = text + tmp
print(text)


def test_random_policy(policy, env):
    _acts = set()
    for i in range(1000):
        act = policy(0)
        _acts.add(act)
        assert env.action_space.contains(act), "Out of the bound!"
    if len(_acts) != 1:
        print(
            "[HINT] Though we call self.policy 'random policy', " \
            "we find that generating action randomly at the beginning " \
            "and then fixing it during updating values period lead to better " \
            "performance. Using purely random policy is not even work! " \
            "We encourage you to investigate this issue."
        )
```

1.2.1 Section 1.1: Make the environment

You need to know

1. How to make an environment
2. How to set the random seed of environment
3. What is observation space and action space

[ ]:
```
# Solve the TODOs and remove `pass`

# [TODO] Just a reminder. Do you add your name and student
# ID in the table at top of the notebook?

# Create the environment
env = gym.make('FrozenLake8x8-v1')

# You need to reset the environment immediately after instantiating env.
env.reset()  # [TODO] uncomment this line
```

```
print("Current observation space: {}".format(env.observation_space))
print("Current action space: {}".format(env.action_space))
print("0 in action space? {}".format(env.action_space.contains(0)))
print("5 in action space? {}".format(env.action_space.contains(5)))
```

```
Current observation space: Discrete(64)
Current action space: Discrete(4)
0 in action space? True
5 in action space? False
```

1.2.2 Section 1.2: Play the environment with random actions

You need to know

1. How to step the environment
2. How to render the environment

[ ]:
```
# Solve the TODOs and remove `pass`

# Run 1000 steps for test, terminate if done.
# You can run this cell multiples times.
env.reset()

while True:
  # take random action
  # [TODO] Uncomment next line
  obs, reward, done, info = env.step(env.action_space.sample())

  # render the environment
  env.render()  # [TODO] Uncomment this line

  print("Current observation: {}\nCurrent reward: {}\n"
        "Whether we are done: {}\ninfo: {}".format(
      obs, reward, done, info
  ))
  wait(sleep=0.5)

  # [TODO] terminate the loop if done
  if done:
    env.reset()
    break
```

```
  (Left)
SFFFFFFF
FFFFFFFF
FFFHFFFF
FFFFFHFF
FFFHFFFF
FHHFFFHF
```

```
FHFFHFHF
FFFHFFFG
Current observation: 35
Current reward: 0.0
Whether we are done: True
info: {'prob': 0.3333333333333333}
```

1.2.3 Section 1.3: Define the evaluation function to value the random baseline

Now we need to define an evaluation function to evaluate a given policy (a function where the input is observation and the output is action).

As a reminder, you should create a `FrozenLake8x8-v1` environment instance by default, reset it after each episode (and at the beginning), step the environment, and terminate episode if done.

After implementing the `evaluate` function, run the next cell to check whether the function is working.

[ ]:
```python
# Solve the TODOs and remove `pass`

def _render_helper(env):
    env.render()
    wait(sleep=0.2)


def evaluate(policy, num_episodes, seed=0, env_name='FrozenLake8x8-v1',␣
 ↪render=False):
    """[TODO] You need to implement this function by yourself. It
    evaluates the given policy and returns the
    average episodic return across #num_episodes episodes.
    We use `seed` argument for testing purpose.
    You should pass the tests in the next cell.

    :param policy: a function whose input is an interger (observation)
    :param num_episodes: number of episodes you wish to run
    :param seed: an interger, used for testing.
    :param env_name: the name of the environment
    :param render: a boolean flag. If true, please call _render_helper
    function.
    :return: the averaged episode reward of the given policy.
    """

    # Create environment (according to env_name, we will use env other than␣
 ↪'FrozenLake8x8-v1')
    env = gym.make(env_name)

    # Seed the environment
    env.seed(seed)
```

```python
    # Build inner loop to run.
    # For each episode, do not set the limit.
    # Only terminate episode (reset environment) when done = True.
    # The episode reward is the sum of all rewards happen within one episode.
    # Call the helper function `render(env)` to render
    rewards = []
    for i in range(num_episodes):
        # reset the environment
        obs = env.reset()
        act = policy(obs)

        ep_reward = 0
        while True:
            # [TODO] run the environment and terminate it if done, collect the
            # reward at each step and sum them to the episode reward.
            obs, reward, done, info = env.step(act)
            act = policy(obs)
            ep_reward += reward
            if done:
                break

        rewards.append(ep_reward)

    return np.mean(rewards)

# [TODO] Run next cell to test your implementation!
```

[ ]:
```python
# Run this cell without modification

# Run this cell to test the correctness of your implementation of `evaluate`.
LEFT = 0
DOWN = 1
RIGHT = 2
UP = 3

def expert(obs):
    """Go down if agent at the right edge, otherwise go right."""
    return DOWN if (obs + 1) % 8 == 0 else RIGHT

def assert_equal(seed, value, env_name):
    ret = evaluate(expert, 1000, seed, env_name=env_name)
    assert ret == value, \
    "When evaluate on seed {}, 1000 episodes, in {} environment, the " \
    "averaged reward should be {}. But you get {}." \
    "".format(seed, env_name, value, ret)

assert_equal(0, 0.065, 'FrozenLake8x8-v1')
```

```
assert_equal(1, 0.059, 'FrozenLake8x8-v1')
assert_equal(2, 0.055, 'FrozenLake8x8-v1')

assert_equal(0, 0.026, 'FrozenLake-v1')
assert_equal(1, 0.034, 'FrozenLake-v1')
assert_equal(2, 0.028, 'FrozenLake-v1')

print("Test Passed!")
print("\nAs a baseline, the mean episode reward of a hand-craft "
      "agent is: ", evaluate(expert, 1000))
```

```
Test Passed!

As a baseline, the mean episode reward of a hand-craft agent is:  0.065
```

Congradiation! You have finished section 1 (if and only if not error happens above).

If you want to do more investigation, feel free to open new cells via pressing `B` after the next cells and write code in it, so that you can reuse some result in this notebook. Remember to write sufficient comments and documents to let others know what you are doing.

[ ]:
```
# You can do more inverstigation here if you wish. Leave it blank if you don't.
```

1.3 Section 2: Model-based Tabular RL

(65/100 points)

We have learned how to use the Gym environment to run an episode, as well as how to interact between the agent (policy) and environment via `env.step(action)` to collect observation, reward, done, and possible extra information.

Now we need to build the basic tabular RL algorithm to solve this environment. **Note that compared to the model-free methods in the Sec.3, the algorithms in this section needs to access the internal information of the environment, namely the transition dynamics**.

In our case, given a state and an action, we need to know which state current environment would jump to, the probability of this transition, and the reward of the transition. You will see that we provide you a helper function `self._get_transitions(state, action)` that takes state and action as input and return you a list of possible transitions.

You will use a class to represent a Trainer, which seems to be over-complex for tabular RL. But we will use the same framework in the future assignments, or even in your future research. So it would be helpful for you to get familiar with how to implement an RL algorithm in a class-orientetd programming style, as a first step toward the implementation of state of the art RL algorithm.

[ ]:
```
# Run this cell without modification

class TabularRLTrainerAbstract:
```

```python
    """This is the abstract class for tabular RL trainer. We will inherent the␣
↪specify
    algorithm's trainer from this abstract class, so that we can reuse the␣
↪codes like
    getting the dynamic of the environment (self._get_transitions()) or␣
↪rendering the
    learned policy (self.render())."""

    def __init__(self, env_name='FrozenLake8x8-v1', model_based=True):
        self.env_name = env_name
        self.env = gym.make(self.env_name)
        self.action_dim = self.env.action_space.n
        self.obs_dim = self.env.observation_space.n

        self.model_based = model_based

    def _get_transitions(self, state, act):
        """Query the environment to get the transition probability,
        reward, the next state, and done given a pair of state and action.
        We implement this function for you. But you need to know the
        return format of this function.
        """
        self._check_env_name()
        assert self.model_based, "You should not use _get_transitions in " \
            "model-free algorithm!"

        # call the internal attribute of the environments.
        # `transitions` is a list contain all possible next states and the
        # probability, reward, and termination indicater corresponding to it
        transitions = self.env.env.P[state][act]

        # Given a certain state and action pair, it is possible
        # to find there exist multiple transitions, since the
        # environment is not deterministic.
        # You need to know the return format of this function: a list of dicts
        ret = []
        for prob, next_state, reward, done in transitions:
            ret.append({
                "prob": prob,
                "next_state": next_state,
                "reward": reward,
                "done": done
            })
        return ret

    def _check_env_name(self):
        assert self.env_name.startswith('FrozenLake')
```

```python
    def print_table(self):
        """print beautiful table, only work for FrozenLake8X8-v1 env. We
        write this function for you."""
        self._check_env_name()
        print_table(self.table)

    def train(self):
        """Conduct one iteration of learning."""
        raise NotImplementedError("You need to override the "
                                  "Trainer.train() function.")

    def evaluate(self):
        """Use the function you write to evaluate current policy.
        Return the mean episode reward of 1000 episodes when seed=0."""
        result = evaluate(self.policy, 1000, env_name=self.env_name)
        return result

    def render(self):
        """Reuse your evaluate function, render current policy
        for one episode when seed=0"""
        evaluate(self.policy, 1, render=True, env_name=self.env_name)
```

1.3.1 Section 2.1: Policy Iteration

Recall the process of policy iteration:

1. Update the state value function, given all possible transitions at current state of the environment.
2. Find the best policy that earns highest value under current state value function.
3. If the best policy is identical to the previous one then stop the training. Otherwise, return to step 1.

In step 1, the way to update the state value function is by

$$v_{k+1} = E_{s'}[r(s,a) + \gamma v_k(s')]$$

wherein the a is given by current policy, s' is next state, r is the reward, $v_k(s')$ is the next state value given by the old (not updated yet) value function. The expectation is computed among all possible transitions (given a state and action pair, it is possible to have many different next states, since the environment is not deterministic).

In step 2, the best policy is the one that takes the action with maximal expected return given a state:

$$a = argmax_a E_{s'}[r(s,a) + \gamma v_k(s')]$$

Policy iteration algorithm has an outer loop (update policy, step 1 to 3) and an inner loop (fit the value function, within step 1).

In each outer loop, we call once `trainer.train()`, where we call `trainer.update_value_function()` once to update the value function (the state value table).

After that we call `trainer.update_policy()` to update the current policy.

`trainer` object has a `trainer.policy` attribute, which is a function that takes observation as input and returns an action.

You should implement the trainer following the framework we already wrote for you. Please carefully go through the codes and finish all `TODO` in it.

[ ]:
```python
# Solve the TODOs and remove `pass`

class PolicyItertaionTrainer(TabularRLTrainerAbstract):
    def __init__(self, gamma=1.0, epsilon=1e-10, env_name='FrozenLake8x8-v1'):
        super(PolicyItertaionTrainer, self).__init__(env_name)

        # discount factor
        self.gamma = gamma

        # value function convergence criterion
        self.epsilon = epsilon

        # build the value table for each possible observation
        self.table = np.zeros((self.obs_dim,))

        # [TODO] you need to implement a random policy at the beginning.
        # It is a function that take an integer (state or say observation)
        # as input and return an interger (action).
        # remember, you can use self.action_dim to get the dimension (range)
        # of the action, which is an integer in range
        # [0, ..., self.action_dim - 1]
        # hint: generating random action at each call of policy may lead to
        #  failure of convergence, try generate random actions at initializtion
        #  and fix it during the training.
        def obs2pol():
            obs2pol = {}

            def random_policy(obs):
                if obs not in obs2pol:
                    obs2pol[obs] = np.random.randint(self.action_dim)
                return obs2pol[obs]

            return random_policy
```

```python
        self.policy = obs2pol()

        # test your random policy
        test_random_policy(self.policy, self.env)

    def train(self):
        """Conduct one iteration of learning."""
        # [TODO] value function may be need to be reset to zeros.
        # if you think it should, than do it. If not, then move on.
        # hint: the value function is equivalent to self.table,
        #   a numpy array with length 64.
        # self.table = np.zeros((self.obs_dim,))

        self.update_value_function()
        self.update_policy()

    def update_value_function(self):
        count = 0  # count the steps of value updates
        while True:
            old_table = self.table.copy()

            for state in range(self.obs_dim):
                action = self.policy(state)
                transition_list = self._get_transitions(state, action)
                state_value = 0

                for transition in transition_list:
                    prob = transition['prob']
                    reward = transition['reward']
                    next_state = transition['next_state']
                    done = transition['done']

                    # [TODO] what is the right state value?
                    # hint: you should use reward, self.gamma, old_table, prob,
                    # and next_state to compute the state value
                    state_value += prob * (reward + self.gamma *␣
↪old_table[next_state])

                # update the state value
                self.table[state] = state_value

            # [TODO] Compare the old_table and current table to
            #  decide whether to break the value update process.
            # hint: you should use self.eps, old_table and self.table
            should_break = np.sum(np.abs(old_table - self.table)) <= self.
↪epsilon
```

```
            if should_break:
                break
            count += 1

            if count % 200 == 0:
                # disable this part if you think debug message annoying.
                print("[DEBUG]\tUpdated values for {} steps. "
                      "Difference between new and old table is: {}".format(
                    count, np.sum(np.abs(old_table - self.table))
                ))
            if count > 4000:
                print("[HINT] Are you sure your codes is OK? It shouldn't be "
                      "so hard to update the value function. You already "
                      "use {} steps to update value function within "
                      "single iteration.".format(count))
            if count > 6000:
                raise ValueError("Clearly your code has problem. Check it!")

    def update_policy(self):
        """You need to define a new policy function, given current
        value function. The best action for a given state is the one that
        has greatest expected return.

        To optimize computing efficiency, we introduce a policy table,
        which take state as index and return the action given a state.
        """
        policy_table = np.zeros([self.obs_dim, ], dtype=np.int64)

        for state in range(self.obs_dim):
            state_action_values = [0] * self.action_dim

            # [TODO] assign the action with greatest "value"
            # to policy_table[state]
            # hint: what is the proper "value" here?
            #  you should use table, gamma, reward, prob,
            #  next_state and self._get_transitions() function
            #  as what we done at self.update_value_function()
            #  Bellman equation may help.
            for action in range(self.action_dim):
                transition_list = self._get_transitions(state, action)

                for transition in transition_list:
                    prob = transition['prob']
                    reward = transition['reward']
                    next_state = transition['next_state']
                    done = transition['done']
```

```
                    state_action_values[action] += prob * (reward + self.gamma␣
↪* self.table[next_state])

                best_action = np.argmax(np.array(state_action_values))

            policy_table[state] = best_action

        self.policy = lambda obs: policy_table[obs]
```

Now we have built the Trainer class for policy iteration algorithm. In the following few cells, we will train the agent to solve the problem and evaluate its performance.

[ ]:
```
# Solve the TODOs and remove `pass`

# Managing configurations of your experiments is important for your research.
default_pi_config = dict(
    max_iteration=1000,
    evaluate_interval=1,
    gamma=1.0,
    epsilon=1e-10
)


def policy_iteration(train_config=None):
    config = default_pi_config.copy()
    if train_config is not None:
        config.update(train_config)

    trainer = PolicyItertaionTrainer(gamma=config['gamma'],␣
↪epsilon=config['epsilon'])

    old_policy_result = {
        obs: -1 for obs in range(trainer.obs_dim)
    }

    for i in range(config['max_iteration']):

        # train the agent
        trainer.train()  # [TODO] please uncomment this line

        # [TODO] compare the new policy with old policy to check whether
        #  should we stop. If new and old policy have same output given any
        #  observation, then we consider the algorithm is converged and
        #  should be stopped.
        new_policy_result = {
            obs: trainer.policy(obs) for obs in range(trainer.obs_dim)
        }
```

```python
        # print(new_policy_result)

        should_stop = (new_policy_result == old_policy_result)

        if should_stop:
            print("We found policy is not changed anymore at "
                  "itertaion {}. Current mean episode reward "
                  "is {}. Stop training.".format(i, trainer.evaluate()))
            break

        # renew the old_policy_result
        old_policy_result = new_policy_result

        # evaluate the result
        if i % config['evaluate_interval'] == 0:
            print(
                "[INFO]\tIn {} iteration, current mean episode reward is {}."
                "".format(i, trainer.evaluate()))

            if i > 20:
                print("You sure your codes is OK? It shouldn't take so many "
                      "({}) iterations to train a policy iteration "
                      "agent.".format(i))

    assert trainer.evaluate() > 0.8, \
        "We expect to get the mean episode reward greater than 0.8. " \
        "But you get: {}. Please check your codes.".format(trainer.evaluate())

    return trainer
```

[ ]:
```python
# Run this cell without modification

# It may be confusing to call a trainer agent. But that's what we normally do.
pi_agent = policy_iteration()
```

```
[DEBUG] Updated values for 200 steps. Difference between new and old table is:
3.6956162779916055e-05
[DEBUG] Updated values for 400 steps. Difference between new and old table is:
1.3648746115369034e-08
[INFO]  In 0 iteration, current mean episode reward is 0.0.
[DEBUG] Updated values for 200 steps. Difference between new and old table is:
0.01235529155800108
[DEBUG] Updated values for 400 steps. Difference between new and old table is:
0.0006563896810581893
[DEBUG] Updated values for 600 steps. Difference between new and old table is:
3.487148897834448e-05
[DEBUG] Updated values for 800 steps. Difference between new and old table is:
```

```
1.8525896713465773e-06
[DEBUG] Updated values for 1000 steps. Difference between new and old table is:
9.842104765553361e-08
[DEBUG] Updated values for 1200 steps. Difference between new and old table is:
5.2287362356517875e-09
[DEBUG] Updated values for 1400 steps. Difference between new and old table is:
2.777825296174097e-10
[INFO]  In 1 iteration, current mean episode reward is 0.0.
[INFO]  In 2 iteration, current mean episode reward is 0.621.
[DEBUG] Updated values for 200 steps. Difference between new and old table is:
2.4654778120754284e-05
[DEBUG] Updated values for 400 steps. Difference between new and old table is:
6.844590012822227e-09
[INFO]  In 3 iteration, current mean episode reward is 0.867.
[DEBUG] Updated values for 200 steps. Difference between new and old table is:
1.93881767239501e-05
[DEBUG] Updated values for 400 steps. Difference between new and old table is:
5.561104632345604e-10
[INFO]  In 4 iteration, current mean episode reward is 0.688.
[DEBUG] Updated values for 200 steps. Difference between new and old table is:
0.00017592727370212228
[DEBUG] Updated values for 400 steps. Difference between new and old table is:
8.922013322643085e-06
[DEBUG] Updated values for 600 steps. Difference between new and old table is:
4.7386343844657564e-07
[DEBUG] Updated values for 800 steps. Difference between new and old table is:
2.51745240625878e-08
[DEBUG] Updated values for 1000 steps. Difference between new and old table is:
1.337427335545982e-09
[INFO]  In 5 iteration, current mean episode reward is 0.87.
[DEBUG] Updated values for 200 steps. Difference between new and old table is:
4.636333325819253e-05
[DEBUG] Updated values for 400 steps. Difference between new and old table is:
2.126145169506488e-07
[DEBUG] Updated values for 600 steps. Difference between new and old table is:
9.75014791126938e-10
[INFO]  In 6 iteration, current mean episode reward is 0.856.
[DEBUG] Updated values for 200 steps. Difference between new and old table is:
6.185307305713039e-05
[DEBUG] Updated values for 400 steps. Difference between new and old table is:
1.2365406895364917e-06
[DEBUG] Updated values for 600 steps. Difference between new and old table is:
2.4720402697075983e-08
[DEBUG] Updated values for 800 steps. Difference between new and old table is:
4.942000836338778e-10
[INFO]  In 7 iteration, current mean episode reward is 0.845.
[INFO]  In 8 iteration, current mean episode reward is 0.853.
We found policy is not changed anymore at itertaion 9. Current mean episode
```

reward is 0.853. Stop training.

[ ]:
```
# Run this cell without modification

print("Your policy iteration agent achieve {} mean episode reward. The optimal␣
↪score "
      "should be closed to {}.".format(pi_agent.evaluate(), 0.86))
```

Your policy iteration agent achieve 0.853 mean episode reward. The optimal score should be closed to 0.86.

[ ]:
```
# Run this cell without modification

pi_agent.render()
```

[ ]:
```
# Run this cell without modification

pi_agent.print_table()
```

```
+-----+-----+-----State Value Mapping-----+-----+-----+
|     |  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |
|-----+-----+-----+-----+-----+-----+-----+-----+-----|
| 0   |1.000|1.000|1.000|1.000|1.000|1.000|1.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   |1.000|1.000|1.000|1.000|1.000|1.000|1.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 2   |1.000|0.978|0.926|0.000|0.857|0.946|0.982|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 3   |1.000|0.935|0.801|0.475|0.624|0.000|0.945|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 4   |1.000|0.826|0.542|0.000|0.539|0.611|0.852|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 5   |1.000|0.000|0.000|0.168|0.383|0.442|0.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 6   |1.000|0.000|0.195|0.121|0.000|0.332|0.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 7   |1.000|0.732|0.463|0.000|0.277|0.555|0.777|0.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Congratulations! You have successfully implemented the policy iteration trainer (if and only if no error happens at the above cells).

Here are few further problems for you to investigate:

1. What is the impact of the discount factor gamma?
2. What is the impact of the value function convergence criterion epsilon?

If you are interested in doing more investigation (not limited to these two), feel free to open new cells at the end of this notebook and left a clear trace of your thinking and coding, which leads to extra credit if you do a good job. It's an optional job, and you can ignore it.

Now let's continue our journey!

1.3.2 Section 2.2: Value Iteration

Recall the idea of value iteration. We update the state value:

$$v_{k+1}(s) = \max_a E_{s'}[r(s,a) + \gamma v_k(s')]$$

wherein the s' is next state, r is the reward, $v_k(s')$ is the next state value given by the old (not updated yet) value function. The expectation is computed among all possible transitions (given a state and action pair, it is possible to have many different next states, since the environment is not deterministic).

The value iteration algorithm does not require an inner loop. It computes the expected return of all possible actions at a given state and uses the maximum of them as the state value. You can imagine it "pretends" we already have the optimal policy and run policy iteration based on it. Therefore we do not need to maintain a policy object in a trainer. We only need to retrieve the optimal policy using the same rule as policy iteration, given current value function.

You should implement the trainer following the framework we already wrote for you. Please carefully go through the code and finish all `TODO` in it.

[ ]:
```python
# Solve the TODOs and remove `pass`

class ValueIterationTrainer(PolicyItertaionTrainer):
    """Note that we inherate Policy Iteration Trainer, to resue the
    code of update_policy(). It's same since it get optimal policy from
    current state-value table (self.table).
    """

    def __init__(self, gamma=1.0, env_name='FrozenLake8x8-v1'):
        super(ValueIterationTrainer, self).__init__(gamma, None, env_name)

    def train(self):
        """Conduct one iteration of learning."""
        # [TODO] value function may be need to be reset to zeros.
        # if you think it should, than do it. If not, then move on.
        # self.table = np.zeros((self.obs_dim,))
```

```
        # In value iteration, we do not explicit require a
        # policy instance to run. We update value function
        # directly based on the transitions. Therefore, we
        # don't need to run self.update_policy() in each step.
        self.update_value_function()

    def update_value_function(self):
        old_table = self.table.copy()

        for state in range(self.obs_dim):
            state_value = 0

            # [TODO] what should be de right state value?
            # hint: try to compute the state_action_values first
            state_action_values = [0] * self.action_dim
            for action in range(self.action_dim):
                transition_list = self._get_transitions(state, action)

                for transition in transition_list:
                    prob = transition['prob']
                    reward = transition['reward']
                    next_state = transition['next_state']
                    done = transition['done']
                    state_action_values[action] += prob * (reward + self.gamma␣
↪* old_table[next_state])

                state_value = np.max(np.array(state_action_values))

            self.table[state] = state_value

        # Till now the one step value update is finished.
        # You can see that we do not use a inner loop to update
        # the value function like what we did in policy iteration.
        # This is because to compute the state value, which is
        # a expectation among all possible action given by a
        # specified policy, we **pretend** already own the optimal
        # policy (the max operation).

    def evaluate(self):
        """Since in value itertaion we do not maintain a policy function,
        so we need to retrieve it when we need it."""
        self.update_policy()
        return super().evaluate()

    def render(self):
        """Since in value itertaion we do not maintain a policy function,
```

```python
        so we need to retrieve it when we need it."""
        self.update_policy()
        return super().render()
```

[ ]:
```python
# Solve the TODOs and remove `pass`

# Managing configurations of your experiments is important for your research.
default_vi_config = dict(
    max_iteration=10000,
    evaluate_interval=100,  # don't need to update policy each iteration
    gamma=1.0,
    epsilon=1e-10
)


def value_iteration(train_config=None):
    config = default_vi_config.copy()
    if train_config is not None:
        config.update(train_config)

    # [TODO] initialize Value Iteration Trainer. Remember to pass
    #  config['gamma'] to it.
    trainer = ValueIterationTrainer(gamma=config['gamma'])

    old_state_value_table = trainer.table.copy()

    old_policy_result = {
        obs: -1 for obs in range(trainer.obs_dim)
    }

    for i in range(config['max_iteration']):

        # train the agent
        trainer.train()  # [TODO] please uncomment this line

        # evaluate the result
        if i % config['evaluate_interval'] == 0:
            print("[INFO]\tIn {} iteration, current "
                  "mean episode reward is {}.".format(
                i, trainer.evaluate()
            ))

            # [TODO] compare the new policy with old policy to check should
            #  we stop.
            # [HINT] If new and old policy have same output given any
            #  observation, then we consider the algorithm is converged and
            #  should be stopped.
```

```
            new_policy_result = {
                obs: trainer.policy(obs) for obs in range(trainer.obs_dim)
            }

            should_stop = (new_policy_result == old_policy_result)

            if should_stop:
                print("We found policy is not changed anymore at "
                      "itertaion {}. Current mean episode reward "
                      "is {}. Stop training.".format(i, trainer.evaluate())) #␣
↪update the policy when stop
                break

            old_policy_result = new_policy_result

            if i > 3000:
                print("You sure your codes is OK? It shouldn't take so many "
                      "({}) iterations to train a policy iteration "
                      "agent.".format(i)
                    )

    assert trainer.evaluate() > 0.8, \
        "We expect to get the mean episode reward greater than 0.8. " \
        "But you get: {}. Please check your codes.".format(trainer.evaluate())

    return trainer
```

[ ]:
```
# Run this cell without modification

vi_agent = value_iteration()
```

```
[INFO]  In 0 iteration, current mean episode reward is 0.0.
[INFO]  In 100 iteration, current mean episode reward is 0.892.
[INFO]  In 200 iteration, current mean episode reward is 0.867.
[INFO]  In 300 iteration, current mean episode reward is 0.867.
[INFO]  In 400 iteration, current mean episode reward is 0.867.
[INFO]  In 500 iteration, current mean episode reward is 0.867.
We found policy is not changed anymore at itertaion 500. Current mean episode
reward is 0.867. Stop training.
```

[ ]:
```
# Run this cell without modification

print("Your value iteration agent achieve {} mean episode reward. The optimal␣
↪score "
      "should be almost {}.".format(vi_agent.evaluate(), 0.86))
```

```
Your value iteration agent achieve 0.867 mean episode reward. The optimal score
```

should be almost 0.86.

[ ]:
```
# Run this cell without modification

vi_agent.render()
```

[ ]:
```
# Run this cell without modification

vi_agent.print_table()
```

```
+-----+-----+-----State Value Mapping-----+-----+-----+
|     |  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |
|-----+-----+-----+-----+-----+-----+-----+-----+-----|
| 0   |0.999|0.999|0.999|0.999|0.999|0.999|0.999|0.999|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   |0.999|0.999|0.999|0.999|0.999|0.999|0.999|0.999|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 2   |0.998|0.976|0.925|0.000|0.856|0.945|0.981|0.999|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 3   |0.997|0.932|0.799|0.474|0.623|0.000|0.944|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 4   |0.997|0.823|0.541|0.000|0.539|0.611|0.851|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 5   |0.996|0.000|0.000|0.168|0.383|0.442|0.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 6   |0.996|0.000|0.194|0.121|0.000|0.332|0.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 7   |0.996|0.728|0.461|0.000|0.277|0.555|0.777|0.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Congratulation! You have successfully implemented the value iteration trainer (if and only if no error happens at the above cells). Few further problems for you to investigate:

1. Do you see that some iteration during training yields better rewards than the final one? Why does that happen?
2. What is the impact of the discount factor gamma?
3. What is the impact of the value function convergence criterion epsilon?

If you are interested in doing more investigation (not limited to these two), feel free to open new cells at the end of this notebook and left a clear trace of your thinking and coding, which leads to extra credit if you do a good job. It's an optional job, and you can ignore it.

Now let's continue our journey!

1.3.3 Section 2.3: Compare two model-based agents

Now we have two agents: `pi_agent` and `vi_agent`. They are believed to be the optimal policy in this environment. Can you compare the policy of two of them and use a clean and clear description or figures to show your conclusion?

[ ]:
```python
# Solve the TODO and remove `pass`

# [TODO] try to compare two trained agents' policies
# hint: trainer.print_table() may give you a better sense.
print("vi_agent:")
vi_agent.print_table()
print("pi_agent:")
pi_agent.print_table()
```

```
vi_agent:
+-----+-----+-----State Value Mapping-----+-----+-----+
|     |  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |
|-----+-----+-----+-----+-----+-----+-----+-----+-----|
| 0   |0.999|0.999|0.999|0.999|0.999|0.999|0.999|0.999|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   |0.999|0.999|0.999|0.999|0.999|0.999|0.999|0.999|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 2   |0.998|0.976|0.925|0.000|0.856|0.945|0.981|0.999|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 3   |0.997|0.932|0.799|0.474|0.623|0.000|0.944|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 4   |0.997|0.823|0.541|0.000|0.539|0.611|0.851|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 5   |0.996|0.000|0.000|0.168|0.383|0.442|0.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 6   |0.996|0.000|0.194|0.121|0.000|0.332|0.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 7   |0.996|0.728|0.461|0.000|0.277|0.555|0.777|0.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+

pi_agent:
```

```
+-----+-----+-----State Value Mapping-----+-----+-----+
|     |   0 |   1 |   2 |   3 |   4 |   5 |   6 |   7 |
|-----+-----+-----+-----+-----+-----+-----+-----+-----|
| 0   |1.000|1.000|1.000|1.000|1.000|1.000|1.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   |1.000|1.000|1.000|1.000|1.000|1.000|1.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 2   |1.000|0.978|0.926|0.000|0.857|0.946|0.982|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 3   |1.000|0.935|0.801|0.475|0.624|0.000|0.945|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 4   |1.000|0.826|0.542|0.000|0.539|0.611|0.852|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 5   |1.000|0.000|0.000|0.168|0.383|0.442|0.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 6   |1.000|0.000|0.195|0.121|0.000|0.332|0.000|1.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 7   |1.000|0.732|0.463|0.000|0.277|0.555|0.777|0.000|
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

I conclude that **vi_agent** is **better** in this scenario

- The mean episode reward of vi_agent is *0.867* and pi_agents is *0.853*.
- For policy iteration, we iterate the value and policy step by step and then get the opitmal policy by the opitmal value. For value iteration, we only get the opitmal policy after iterating the value and getting the opitmal value.
- According to the iteration logs, we can see that vi_agent has *small variance* and the pi_agent has large variance.

[ ]:
```python
# Test our trained trainer based on policy iteration
obs = env.reset()

while True:
    # take agent action
    # [TODO] Uncomment next line
    obs, reward, done, info = env.step(pi_agent.policy(obs))

    # render the environment
```

```python
    env.render()  # [TODO] Uncomment this line

    print("Current observation: {}\nCurrent reward: {}\n"
          "Whether we are done: {}\ninfo: {}".format(
        obs, reward, done, info
    ))
    wait(sleep=0.5)

    # [TODO] terminate the loop if done
    if done:
        env.reset()
        break
```

```
  (Right)
SFFFFFFF
FFFFFFFF
FFFHFFFF
FFFFFHFF
FFFHFFFF
FHHFFFHF
FHFFHFHF
FFFHFFFG
Current observation: 63
Current reward: 1.0
Whether we are done: True
info: {'prob': 0.3333333333333333}
```

[ ]:
```python
# You can do more inverstigation here if you wish. Leave it blank if you don't.

# Test our trained trainer based on value iteration
obs = env.reset()

while True:
    # take agent action
    obs, reward, done, info = env.step(vi_agent.policy(obs))

    # render the environment
    env.render()

    print("Current observation: {}\nCurrent reward: {}\n"
          "Whether we are done: {}\ninfo: {}".format(
        obs, reward, done, info
    ))
    wait(sleep=0.5)

    if done:
        env.reset()
```

```
        break
```

```
  (Right)
SFFFFFFF
FFFFFFFF
FFFHFFFF
FFFFFHFF
FFFHFFFF
FHHFFFHF
FHFFHFHF
FFFHFFFG
Current observation: 63
Current reward: 1.0
Whether we are done: True
info: {'prob': 0.3333333333333333}
```

We did it!

1.4 Conclusion and Discussion

In this assignment, we learn how to use Gym package, how to use Object Oriented Programming idea to build a basic tabular RL algorithm.

It's OK to leave the following cells empty. In the next markdown cell, you can write whatever you like. Like the suggestion on the course, the confusing problems in the assignments, and so on.

Following the submission instruction in the assignment to submit your assignment to our staff. Thank you!

…