assignment2

October 21, 2021

1 IERG 5350 Assignment 2: Model-free Tabular RL

2021-2022 1st term, IERG 5350: Reinforcement Learning. Department of Information Engineering, The Chinese University of Hong Kong. Course Instructor: Professor ZHOU Bolei. Assignment author: PENG Zhenghao.

Student Name	Student ID
Junda Huang	1155160619

Welecome to the assignment 2 of our RL course. The objective of this assignment is for you to understand the classic methods used in tabular reinforcement learning.

You need to go through this self-contained notebook, which contains dozens of **TODOs** in part of the cells and has special `[TODO]` signs. You need to finish all TODOs.

Please report any code bugs to us via Github issues.

Before you get start, remember to follow the instruction at https://github.com/cuhkrlcourse/ierg5350-assignment-2021 to setup your environment.

1.1 Section 1: SARSA

(30/100 points)

You have noticed that in Assignment 1 - Section 2, we always use the function `trainer._get_transitions()` to get the transition dynamics of the environment, while never call `trainer.env.step()` to really interact with the environment by applying actions. We need to access the internal dynamics of the environment and have somebody implement `_get_transitions` for us.

However, this is not feasible in many cases, especially in some real-world tasks like autonomous driving where the transition dynamics is unknown.

In this section, we will introduce the model-free family of algorithms that do not require to know the transitions: they only get information from `env.step(action)` and collect information by interacting with the environment.

We will continue to use the `TabularRLTrainerAbstract` class to implement algorithms, but remember you should not call `trainer._get_transitions()` anymore.

We will use a simpler environment `FrozenLakerNotSlippery-v0` to conduct experiments, which has a `4 X 4` grids and is deterministic. This is because, in a model-free setting, it's extremely hard for a random agent to achieve the goal for the first time. To reduce the time of experiments, we choose to use a simpler environment. In the bonus section, you can try out model-free RL on `FrozenLake8x8-v1` to see what will happen.

Now go through each section and start your coding!

Recall the idea of SARSA: it's an on-policy TD control method, which has distinct features compared to policy iteration and value iteration methods in the training process:

1. It maintains a state-action pair value function $Q(s_t, a_t) = E\sum_{i=0}\gamma^{t+i}r_{t+i}$ to approximate the Q value.
2. It does not require to know the internal dynamics of the environment.
3. It use an epsilon-greedy strategy to balance exploration and exploitation.

In SARSA algorithm, we update the Q value via TD error:

$$TD(s_t, a_t) = r(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t),$$

wherein we run the policy to get the next action $a_{t+1} = Policy(s_{t+1})$. That's why we call SARSA an on-policy algorithm, since it use the current policy to evaluate Q value.

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha TD(s_t, a_t),$$

wherein α is the learning rate, a hyper-parameter provided by the user.

Now please go through the codes.

[1]:
```
# Run this cell without modification

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

[2]:
```
# 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
    evaluate the given policy and return the mean episode reward.
    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-v0')
    env = gym.make(env_name)

    # Seed the environment
    env.seed(seed)

    # 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.
            ep_obs, reward, ep_done, ep_info = env.step(act)
            ep_reward += reward
            if ep_done:
                break
            act = policy(ep_obs)

        rewards.append(ep_reward)
    if render:
        env.render()
    return np.mean(rewards)

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

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

class TabularRLTrainerAbstract:
    """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')

    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)
```

[4]:

```
# Solve the TODOs and remove `pass`

class SARSATrainer(TabularRLTrainerAbstract):
    def __init__(self,
                 gamma=1.0,
                 eps=0.1,
                 learning_rate=1.0,
                 max_episode_length=100,
                 env_name='FrozenLake8x8-v1'
                 ):
        super(SARSATrainer, self).__init__(env_name, model_based=False)

        # discount factor
        self.gamma = gamma

        # epsilon-greedy exploration policy parameter
        self.eps = eps

        # maximum steps in single episode
        self.max_episode_length = max_episode_length
```

```python
        # the learning rate
        self.learning_rate = learning_rate

        # build the Q table
        # [TODO] uncomment the next line, pay attention to the shape
        self.table = np.zeros((self.obs_dim, self.action_dim))

    def policy(self, obs):
        """Implement epsilon-greedy policy

        It is a function that take an integer (state / observation)
        as input and return an interger (action).
        """

        # [TODO] You need to implement the epsilon-greedy policy here.
        # hint: We have self.eps probability to choose a unifomly random
        #  action in range [0, 1, .., self.action_dim - 1],
        #  otherwise choose action that maximize the Q value
        if np.random.rand() < self.eps:
            return np.random.randint(0, self.action_dim)
        else:
            return np.argmax(self.table[obs])

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

        # No, we should do nothing.

        obs = self.env.reset()
        for t in range(self.max_episode_length):
            act = self.policy(obs)

            next_obs, reward, done, _ = self.env.step(act)
            next_act = self.policy(next_obs)

            # [TODO] compute the TD error, based on the next observation and
            #  action.
            td_error = reward + self.gamma*self.table[next_obs][next_act] - \
                       self.table[obs][act]

            # [TODO] compute the new Q value
            # hint: use TD error, self.learning_rate and old Q value
            new_value = self.table[obs][act] + self.learning_rate*td_error

            self.table[obs][act] = new_value
```

```
            # [TODO] Implement (1) break if done. (2) update obs for next
            #  self.policy(obs) call
            if done:
                break
            obs = next_obs

# [TODO] run the next cell to check your code
```

Now you have finished the SARSA trainer. To make sure your implementation of epsilon-greedy strategy is correct, please run the next cell.

[5]:
```
# Run this cell without modification

# set eps = 0 to disable exploration.
test_trainer = SARSATrainer(eps=0.0)
test_trainer.table.fill(0)

# set the Q value of (obs 0, act 3) to 100, so that it should be taken by
# policy.
test_obs = 0
test_act = test_trainer.action_dim - 1
test_trainer.table[test_obs][test_act] = 100

# assertion
assert test_trainer.policy(test_obs) == test_act, \
    "Your action is wrong! Should be {} but get {}.".format(
        test_act, test_trainer.policy(test_obs))

# delete trainer
del test_trainer

# set eps = 0 to disable exploitation.
test_trainer = SARSATrainer(eps=1.0)
test_trainer.table.fill(0)

act_set = set()
for i in range(100):
    act_set.add(test_trainer.policy(0))

# assertion
assert len(act_set) > 1, ("You sure your uniformaly action selection mechanism"
                          " is working? You only take action {} when "
                          "observation is 0, though we run trainer.policy() "
                          "for 100 times.".format(act_set))
# delete trainer
del test_trainer
```

```
print("Policy Test passed!")
```

```
Policy Test passed!
```

Now run the next cells to see the result.

Note that we use the non-slippery version of a small frozen lake environment `FrozenLakeNotSlipppery-v0` (this is not a ready Gym environment, see `utils.py` for details). This is because, in the model-free setting, it's extremely hard to access the goal for the first time (you should already know that if you watch the agent randomly acting in Assignment 1 - Section 1).

[6]:
```
# Solve TODO

# Managing configurations of your experiments is important for your research.
default_sarsa_config = dict(
    max_iteration=20000,
    max_episode_length=200,
    learning_rate=0.01,
    evaluate_interval=1000,
    gamma=0.8,
    eps=0.3,
    env_name='FrozenLakeNotSlippery-v0'
)



def sarsa(train_config=None):
    config = default_sarsa_config.copy()
    if train_config is not None:
        config.update(train_config)

    trainer = SARSATrainer(
        gamma=config['gamma'],
        eps=config['eps'],
        learning_rate=config['learning_rate'],
        max_episode_length=config['max_episode_length'],
        env_name=config['env_name']
    )

    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 {}."
```

```python
                 "".format(i, trainer.evaluate()))

    if trainer.evaluate() < 0.6:
        print("We expect to get the mean episode reward greater than 0.6. " \
        "But you get: {}. Please check your codes.".format(trainer.evaluate()))

    return trainer
```

[7]:
```python
# Run this cell without modification

sarsa_trainer = sarsa()
```

```
[INFO]  In 0 iteration, current mean episode reward is 0.001.
[INFO]  In 1000 iteration, current mean episode reward is 0.01.
[INFO]  In 2000 iteration, current mean episode reward is 0.656.
[INFO]  In 3000 iteration, current mean episode reward is 0.638.
[INFO]  In 4000 iteration, current mean episode reward is 0.663.
[INFO]  In 5000 iteration, current mean episode reward is 0.639.
[INFO]  In 6000 iteration, current mean episode reward is 0.637.
[INFO]  In 7000 iteration, current mean episode reward is 0.648.
[INFO]  In 8000 iteration, current mean episode reward is 0.666.
[INFO]  In 9000 iteration, current mean episode reward is 0.663.
[INFO]  In 10000 iteration, current mean episode reward is 0.644.
[INFO]  In 11000 iteration, current mean episode reward is 0.65.
[INFO]  In 12000 iteration, current mean episode reward is 0.661.
[INFO]  In 13000 iteration, current mean episode reward is 0.647.
[INFO]  In 14000 iteration, current mean episode reward is 0.636.
[INFO]  In 15000 iteration, current mean episode reward is 0.661.
[INFO]  In 16000 iteration, current mean episode reward is 0.652.
[INFO]  In 17000 iteration, current mean episode reward is 0.655.
[INFO]  In 18000 iteration, current mean episode reward is 0.675.
[INFO]  In 19000 iteration, current mean episode reward is 0.67.
```

[8]:
```python
# Run this cell without modification

sarsa_trainer.print_table()
```

```
=== The state value for action 0 ===
+-----+-----+-----+-----+-----+
|     |  0  |  1  |  2  |  3  |
|-----+-----+-----+-----+-----+
| 0   |0.122|0.124|0.037|0.011|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.173|0.000|0.000|0.000|
|     |     |     |     |     |
```

```
+-----+-----+-----+-----+-----+
| 2   |0.243|0.243|0.352|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.000|0.501|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+


=== The state value for action 1 ===
+-----+-----+-----+-----+-----+
|     |   0 |   1 |   2 |   3 |
|-----+-----+-----+-----+-----+
| 0   |0.170|0.000|0.239|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.248|0.000|0.476|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.000|0.511|0.716|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.498|0.730|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+


=== The state value for action 2 ===
+-----+-----+-----+-----+-----+
|     |   0 |   1 |   2 |   3 |
|-----+-----+-----+-----+-----+
| 0   |0.084|0.071|0.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.000|0.000|0.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.359|0.489|0.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.737|1.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+


=== The state value for action 3 ===
+-----+-----+-----+-----+-----+
|     |   0 |   1 |   2 |   3 |
```

```
|-----+-----+-----+-----+-----+
| 0   |0.122|0.065|0.014|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.124|0.000|0.041|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.169|0.000|0.304|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.352|0.477|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
```

[9]:
```python
# Run this cell without modification

sarsa_trainer.render()
```

```
  (Right)
SFFF
FHFH
FFFH
HFFG
```

Now you have finished the SARSA algorithm.

1.2 Section 2: Q-Learning

(30/100 points)

Q-learning is an off-policy algorithm who differs from SARSA in the computing of TD error. Instead of running policy to get `next_act` a' and get the TD error by:

$r + \gamma Q(s', a') - Q(s, a), a' \sim \pi(\cdot|s')$,

in Q-learning we compute the TD error via:

$r + \gamma \max_{a'} Q(s', a') - Q(s, a)$.

The reason we call it "off-policy" is that the policy involves the computing of next-Q value is not the "behavior policy", instead, it is a "optimal policy" that always takes the best action given current Q values.

[10]:
```python
# Solve the TODOs and remove `pass`

class QLearningTrainer(TabularRLTrainerAbstract):
    def __init__(self,
                 gamma=1.0,
                 eps=0.1,
                 learning_rate=1.0,
```

```
                 max_episode_length=100,
                 env_name='FrozenLake8x8-v1'
                 ):
        super(QLearningTrainer, self).__init__(env_name, model_based=False)
        self.gamma = gamma
        self.eps = eps
        self.max_episode_length = max_episode_length
        self.learning_rate = learning_rate

        # build the Q table
        self.table = np.zeros((self.obs_dim, self.action_dim))

    def policy(self, obs):
        """Implement epsilon-greedy policy

        It is a function that take an integer (state / observation)
        as input and return an interger (action).
        """

        # [TODO] You need to implement the epsilon-greedy policy here.
        # hint: Just copy your codes in SARSATrainer.policy()
        if np.random.random() < self.eps:
            return np.random.randint(0, self.action_dim)
        else:
            return np.argmax(self.table[obs])

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

        # No, we should do nothing.

        obs = self.env.reset()
        for t in range(self.max_episode_length):
            act = self.policy(obs)

            next_obs, reward, done, _ = self.env.step(act)

            # [TODO] compute the TD error, based on the next observation
            # hint: we do not need next_act anymore.
            td_error = reward + self.gamma*np.amax(self.table[next_obs]) - \
                self.table[obs][act]

            # [TODO] compute the new Q value
            # hint: use TD error, self.learning_rate and old Q value
            new_value = self.table[obs][act] + self.learning_rate*td_error
```

```
            self.table[obs][act] = new_value
            obs = next_obs
            if done:
                break
```

[11]:

```
# Solve the TODO

# Managing configurations of your experiments is important for your research.
default_q_learning_config = dict(
    max_iteration=20000,
    max_episode_length=200,
    learning_rate=0.01,
    evaluate_interval=1000,
    gamma=0.8,
    eps=0.3,
    env_name='FrozenLakeNotSlippery-v0'
)


def q_learning(train_config=None):
    config = default_q_learning_config.copy()
    if train_config is not None:
        config.update(train_config)

    trainer = QLearningTrainer(
        gamma=config['gamma'],
        eps=config['eps'],
        learning_rate=config['learning_rate'],
        max_episode_length=config['max_episode_length'],
        env_name=config['env_name']
    )

    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()))

    if trainer.evaluate() < 0.6:
        print("We expect to get the mean episode reward greater than 0.6. " \
        "But you get: {}. Please check your codes.".format(trainer.evaluate()))
```

```python
    return trainer
```

[12]:
```python
# Run this cell without modification

q_learning_trainer = q_learning()
```

```
[INFO]  In 0 iteration, current mean episode reward is 0.0.
[INFO]  In 1000 iteration, current mean episode reward is 0.0.
[INFO]  In 2000 iteration, current mean episode reward is 0.674.
[INFO]  In 3000 iteration, current mean episode reward is 0.645.
[INFO]  In 4000 iteration, current mean episode reward is 0.67.
[INFO]  In 5000 iteration, current mean episode reward is 0.652.
[INFO]  In 6000 iteration, current mean episode reward is 0.645.
[INFO]  In 7000 iteration, current mean episode reward is 0.672.
[INFO]  In 8000 iteration, current mean episode reward is 0.632.
[INFO]  In 9000 iteration, current mean episode reward is 0.668.
[INFO]  In 10000 iteration, current mean episode reward is 0.681.
[INFO]  In 11000 iteration, current mean episode reward is 0.667.
[INFO]  In 12000 iteration, current mean episode reward is 0.661.
[INFO]  In 13000 iteration, current mean episode reward is 0.671.
[INFO]  In 14000 iteration, current mean episode reward is 0.691.
[INFO]  In 15000 iteration, current mean episode reward is 0.676.
[INFO]  In 16000 iteration, current mean episode reward is 0.661.
[INFO]  In 17000 iteration, current mean episode reward is 0.647.
[INFO]  In 18000 iteration, current mean episode reward is 0.663.
[INFO]  In 19000 iteration, current mean episode reward is 0.71.
```

[13]:
```python
# Run this cell without modification

q_learning_trainer.print_table()
```

```
=== The state value for action 0 ===
+-----+-----+-----+-----+-----+
|     |  0  |  1  |  2  |  3  |
|-----+-----+-----+-----+-----+
| 0   |0.262|0.262|0.175|0.013|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.328|0.000|0.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.410|0.410|0.437|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.000|0.640|0.000|
|     |     |     |     |     |
```

```
+-----+-----+-----+-----+-----+


=== The state value for action 1 ===
+-----+-----+-----+-----+-----+
|     |   0 |   1 |   2 |   3 |
|-----+-----+-----+-----+-----+
| 0   |0.328|0.000|0.042|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.410|0.000|0.518|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.000|0.640|0.800|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.640|0.800|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+


=== The state value for action 2 ===
+-----+-----+-----+-----+-----+
|     |   0 |   1 |   2 |   3 |
|-----+-----+-----+-----+-----+
| 0   |0.210|0.088|0.001|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.000|0.000|0.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.512|0.640|0.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.800|1.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+


=== The state value for action 3 ===
+-----+-----+-----+-----+-----+
|     |   0 |   1 |   2 |   3 |
|-----+-----+-----+-----+-----+
| 0   |0.262|0.165|0.011|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.262|0.000|0.023|0.000|
|     |     |     |     |     |
```

```
+-----+-----+-----+-----+-----+
| 2   |0.328|0.000|0.282|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.512|0.640|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
```

[14]:
```
# Run this cell without modification

q_learning_trainer.render()
```

```
  (Right)
SFFF
FHFH
FFFH
HFFG
```

Now you have finished Q-Learning algorithm.

1.3 Section 3: Monte Carlo Control

(40/100 points)

In sections 1 and 2, we implement the on-policy and off-policy versions of the TD Learning algorithms. In this section, we will play with another branch of the model-free algorithm: Monte Carlo Control. You can refer to the 5.3 Monte Carlo Control section of the textbook "Reinforcement Learning: An Introduction" to learn the details of MC control.

The basic idea of MC control is to compute the Q value (state-action value) directly from an episode, without using TD to fit the Q function.

Concretely, we maintain a batch of lists (the total number of lists is `obs_dim * action_dim`), each elememnt of the batch is a list correspondent to a state-action pair. The list is used to store the previously happenning "return" of each state action pair. The "return" here is the discounted accumulative reward of the trajectories starting from the state-action pair: $Return(s_t, a_t) = \sum_{i=0} \gamma^{t+i} r_{t+i}$.

For example, the batch might looks like:

```
[(state="in left upper corner", action="turn right") = [10.0, 20.0, 30.0],
 (state=..., action=...) = [previously recorded return ...],
...
]
```

We will use a dict `self.returns` to store all lists. The keys of the dict are tuples `(obs, act)` and the value of the dict `self.returns[(obs, act)]` is the list to store all returns of the trajectories that starts from `(obs, act)`.

The key point of MC Control method is that we take the mean of this list (the mean of all previous returns) as the Q value of the corresponding state-action pair. In short, MC Control method uses a new way to estimate the values of state-action pairs.

[15]:

```python
# Solve the TODOs and remove `pass`

class MCControlTrainer(TabularRLTrainerAbstract):
    def __init__(self,
                 gamma=1.0,
                 eps=0.3,
                 max_episode_length=100,
                 env_name='FrozenLake8x8-v1'
                 ):
        super(MCControlTrainer, self).__init__(env_name, model_based=False)
        self.gamma = gamma
        self.eps = eps
        self.max_episode_length = max_episode_length

        # build the dict of lists
        self.returns = {}
        for obs in range(self.obs_dim):
            for act in range(self.action_dim):
                self.returns[(obs, act)] = []

        # build the Q table
        self.table = np.zeros((self.obs_dim, self.action_dim))

    def policy(self, obs):
        """Implement epsilon-greedy policy

        It is a function that take an integer (state / observation)
        as input and return an interger (action).
        """

        # [TODO] You need to implement the epsilon-greedy policy here.
        # hint: Just copy your codes in SARSATrainer.policy()
        if np.random.random() < self.eps:
            return np.random.randint(0, self.action_dim)
        else:
            return np.argmax(self.table[obs])

    def train(self):
        """Conduct one iteration of learning."""
        observations = []
        actions = []
        rewards = []

        # [TODO] rollout for one episode, store data in three lists create
        #  above.
        # hint: we do not need to store next observation.
        obs = self.env.reset()
```

```
        for t in range(self.max_episode_length):
            act = self.policy(obs)
            next_obs, reward, done, _ = self.env.step(act)
            observations.append(obs)
            rewards.append(reward)
            actions.append(act)

            if done == True:
                break

            obs = next_obs

        assert len(actions) == len(observations)
        assert len(actions) == len(rewards)

        occured_state_action_pair = set()
        length = len(actions)
        value = 0
        for i in reversed(range(length)):
            # if length = 10, then i = 9, 8, ..., 0

            obs = observations[i]
            act = actions[i]
            reward = rewards[i]

            # [TODO] compute the value reversely
            # hint: value(t) = gamma * value(t+1) + r(t)
            value = self.gamma*value + reward

            if (obs, act) not in occured_state_action_pair:
                occured_state_action_pair.add((obs, act))

                # [TODO] append current return (value) to dict
                # hint: `value` represents the future return due to
                #  current (obs, act), so we need to store this value
                #  in trainer.returns
                self.returns[(obs,act)].append(value)

                # [TODO] compute the Q value from self.returns and write it
                #  into self.table
                self.table[obs][act] = np.mean(self.returns[(obs,act)])

                # we don't need to update the policy since it is
                # automatically adjusted with self.table
```

[16]:
```
# Run this cell without modification
```

```python
# Managing configurations of your experiments is important for your research.
default_mc_control_config = dict(
    max_iteration=20000,
    max_episode_length=200,
    evaluate_interval=1000,
    gamma=0.8,
    eps=0.3,
    env_name='FrozenLakeNotSlippery-v0'
)


def mc_control(train_config=None):
    config = default_mc_control_config.copy()
    if train_config is not None:
        config.update(train_config)

    trainer = MCControlTrainer(
        gamma=config['gamma'],
        eps=config['eps'],
        max_episode_length=config['max_episode_length'],
        env_name=config['env_name']
    )

    for i in range(config['max_iteration']):
        # train the agent
        trainer.train()

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

    if trainer.evaluate() < 0.6:
        print("We expect to get the mean episode reward greater than 0.6. " \
        "But you get: {}. Please check your codes.".format(trainer.evaluate()))

    return trainer
```

[17]:
```python
# Run this cell without modification

mc_control_trainer = mc_control()

sarsa_trainer = sarsa()
```

```
[INFO]  In 0 iteration, current mean episode reward is 0.0.
[INFO]  In 1000 iteration, current mean episode reward is 0.002.
```

```
[INFO]  In 2000 iteration, current mean episode reward is 0.0.
[INFO]  In 3000 iteration, current mean episode reward is 0.0.
[INFO]  In 4000 iteration, current mean episode reward is 0.577.
[INFO]  In 5000 iteration, current mean episode reward is 0.657.
[INFO]  In 6000 iteration, current mean episode reward is 0.681.
[INFO]  In 7000 iteration, current mean episode reward is 0.675.
[INFO]  In 8000 iteration, current mean episode reward is 0.64.
[INFO]  In 9000 iteration, current mean episode reward is 0.665.
[INFO]  In 10000 iteration, current mean episode reward is 0.648.
[INFO]  In 11000 iteration, current mean episode reward is 0.658.
[INFO]  In 12000 iteration, current mean episode reward is 0.654.
[INFO]  In 13000 iteration, current mean episode reward is 0.654.
[INFO]  In 14000 iteration, current mean episode reward is 0.665.
[INFO]  In 15000 iteration, current mean episode reward is 0.637.
[INFO]  In 16000 iteration, current mean episode reward is 0.683.
[INFO]  In 17000 iteration, current mean episode reward is 0.631.
[INFO]  In 18000 iteration, current mean episode reward is 0.658.
[INFO]  In 19000 iteration, current mean episode reward is 0.646.
[INFO]  In 0 iteration, current mean episode reward is 0.0.
[INFO]  In 1000 iteration, current mean episode reward is 0.0.
[INFO]  In 2000 iteration, current mean episode reward is 0.0.
[INFO]  In 3000 iteration, current mean episode reward is 0.0.
[INFO]  In 4000 iteration, current mean episode reward is 0.001.
[INFO]  In 5000 iteration, current mean episode reward is 0.0.
[INFO]  In 6000 iteration, current mean episode reward is 0.0.
[INFO]  In 7000 iteration, current mean episode reward is 0.0.
[INFO]  In 8000 iteration, current mean episode reward is 0.001.
[INFO]  In 9000 iteration, current mean episode reward is 0.0.
[INFO]  In 10000 iteration, current mean episode reward is 0.001.
[INFO]  In 11000 iteration, current mean episode reward is 0.0.
[INFO]  In 12000 iteration, current mean episode reward is 0.0.
[INFO]  In 13000 iteration, current mean episode reward is 0.0.
[INFO]  In 14000 iteration, current mean episode reward is 0.0.
[INFO]  In 15000 iteration, current mean episode reward is 0.0.
[INFO]  In 16000 iteration, current mean episode reward is 0.0.
[INFO]  In 17000 iteration, current mean episode reward is 0.0.
[INFO]  In 18000 iteration, current mean episode reward is 0.0.
[INFO]  In 19000 iteration, current mean episode reward is 0.0.
We expect to get the mean episode reward greater than 0.6. But you get: 0.002.
Please check your codes.
```

[18]:

```
# Run this cell without modification

mc_control_trainer.print_table()
```

```
=== The state value for action 0 ===
+-----+-----+-----+-----+-----+
```

```
|     |   0 |   1 |   2 |   3 |
|-----+-----+-----+-----+-----+
| 0   |0.037|0.010|0.047|0.178|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.048|0.000|0.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.106|0.152|0.336|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.000|0.498|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+


=== The state value for action 1 ===
+-----+-----+-----+-----+-----+
|     |   0 |   1 |   2 |   3 |
|-----+-----+-----+-----+-----+
| 0   |0.137|0.000|0.326|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.219|0.000|0.503|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.000|0.439|0.750|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.457|0.737|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+


=== The state value for action 2 ===
+-----+-----+-----+-----+-----+
|     |   0 |   1 |   2 |   3 |
|-----+-----+-----+-----+-----+
| 0   |0.061|0.179|0.114|0.149|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.000|0.000|0.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.337|0.503|0.000|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.746|1.000|0.000|
```

```
|     |     |     |     |     |
+-----+-----+-----+-----+-----+


=== The state value for action 3 ===
+-----+-----+-----+-----+-----+
|     |   0 |   1 |   2 |   3 |
|-----+-----+-----+-----+-----+
| 0   |0.052|0.037|0.198|0.151|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 1   |0.063|0.000|0.242|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 2   |0.104|0.000|0.324|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
| 3   |0.000|0.331|0.504|0.000|
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
```

[19]:

```
# Run this cell without modification

mc_control_trainer.render()
```

```
  (Right)
SFFF
FHFH
FFFH
HFFG
```

1.4 Secion 4 Bonus (optional): Tune and train FrozenLake8x8-v1 with Model-free algorithms

You have noticed that we use a simpler environment `FrozenLakeNotSlippery-v0` which has only 16 states and is not stochastic. Can you try to train Model-free families of algorithm using the `FrozenLake8x8-v1` environment? Tune the hyperparameters and compare the results between different algorithms.

Hint: It's not easy to train model-free algorithm in `FrozenLake8x8-v1`. Failure is excepted.

[23]:

```
# It's ok to leave this cell commented.

new_config = dict(
    max_iteration = 320000,
    max_episode_length = 1000,
    learning_rate = 0.0001,
```

```
    evaluate_interval = 8000,
    gamma = 0.9,
    eps = 0.6,
    env_name="FrozenLake8x8-v1"
)

new_mc_control_trainer = mc_control(new_config)
new_q_learning_trainer = q_learning(new_config)
new_sarsa_trainer = sarsa(new_config)
```

```
[INFO]  In 0 iteration, current mean episode reward is 0.0.
[INFO]  In 8000 iteration, current mean episode reward is 0.0.
[INFO]  In 16000 iteration, current mean episode reward is 0.003.
[INFO]  In 24000 iteration, current mean episode reward is 0.014.
[INFO]  In 32000 iteration, current mean episode reward is 0.018.
[INFO]  In 40000 iteration, current mean episode reward is 0.02.
[INFO]  In 48000 iteration, current mean episode reward is 0.02.
[INFO]  In 56000 iteration, current mean episode reward is 0.019.
[INFO]  In 64000 iteration, current mean episode reward is 0.011.
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/tmp/ipykernel_17601/733399183.py in <module>
     11 )
     12
---> 13 new_mc_control_trainer = mc_control(new_config)
     14 new_q_learning_trainer = q_learning(new_config)
     15 new_sarsa_trainer = sarsa(new_config)

/tmp/ipykernel_17601/567041371.py in mc_control(train_config)
     26     for i in range(config['max_iteration']):
     27         # train the agent
---> 28         trainer.train()
     29
     30         # evaluate the result

/tmp/ipykernel_17601/3356358659.py in train(self)
     86                 # [TODO] compute the Q value from self.returns and write␣
 ↪it
     87                 #  into self.table
---> 88                 self.table[obs][act] = np.mean(self.returns[(obs,act)])
     89
     90                 # we don't need to update the policy since it is

<__array_function__ internals> in mean(*args, **kwargs)
```

```
~/anaconda3/envs/ierg5350/lib/python3.7/site-packages/numpy/core/fromnumeric.py
↪in mean(a, axis, dtype, out, keepdims)
   3371
   3372     return _methods._mean(a, axis=axis, dtype=dtype,
-> 3373                           out=out, **kwargs)

   3374
   3375

~/anaconda3/envs/ierg5350/lib/python3.7/site-packages/numpy/core/_methods.py in
↪_mean(a, axis, dtype, out, keepdims)
    142
    143 def _mean(a, axis=None, dtype=None, out=None, keepdims=False):
--> 144     arr = asanyarray(a)
    145
    146     is_float16_result = False

~/anaconda3/envs/ierg5350/lib/python3.7/site-packages/numpy/core/_asarray.py in
↪asanyarray(a, dtype, order)
    134
    135     """
--> 136     return array(a, dtype, copy=False, order=order, subok=True)
    137
    138

KeyboardInterrupt:
```

Now you have implement the MC Control algorithm. You have finished this section. If you want to do more investigation like comparing the policy provided by SARSA, Q-Learning and MC Control, then you can do it in the next cells. It's OK to leave it blank.

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

1.5 Conclusion and Discussion

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!

.

[ ]:
```

```