Assignment 2

ECS7002P - Artificial Intelligence in Games

November 9, 2020

In this assignment, you will implement a variety of reinforcement learning algorithms to find policies for the *frozen lake* environment. Please read this entire document before you start working on the assignment.

1 Environment

The frozen lake environment has two main variants: the small frozen lake (Fig. 1) and the big frozen lake (Fig. 2). In both cases, each *tile* in a square grid corresponds to a state. There is also an additional *absorbing state*, which will be introduced soon. There are four types of tiles: start (grey), frozen lake (light blue), hole (dark blue), and goal (white). The agent has four actions, which correspond to moving one tile up, left, down, or right. However, with probability 0.1, the environment ignores the desired direction and the agent *slips* (moves one tile in a random direction). An action that would cause the agent to move outside the grid leaves the state unchanged.

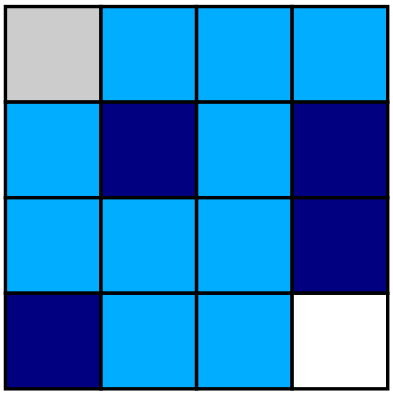

Figure 1: Small frozen lake

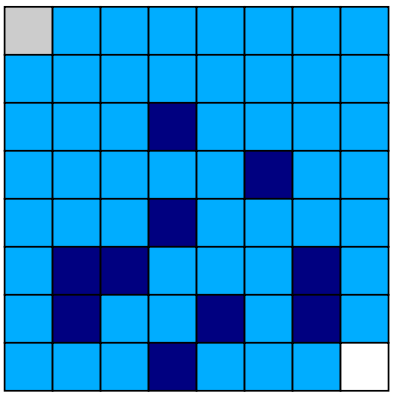

Figure 2: Big frozen lake

The agent receives reward 1 upon taking an action at the goal. In every other case, the agent receives zero reward. Note that the agent does not receive a reward upon moving *into* the goal (nor a negative reward upon moving into a hole). Upon taking an action at the goal or in a hole, the agent moves into the absorbing state. Every action taken at the absorbing state leads to the absorbing state, which also does not provide rewards. Assume a discount factor of $\gamma = 0.9$.

For the purposes of model-free reinforcement learning (or interactive testing), the agent is able to interact with the frozen lake for a number of time steps that is equal to the number of tiles.

Your first task is to implement the frozen lake environment. Using either Python or Java, try to mimic the Python interface presented in Listing 1.

Listing 1: Frozen lake environment.

```
import numpy as np
import contextlib

# Configures numpy print options
@contextlib.contextmanager
def _printoptions(*args, **kwargs):
    original = np.get_printoptions()
    np.set_printoptions(*args, **kwargs)
    try:
        yield
    finally:
        np.set_printoptions(**original)


class EnvironmentModel:
    def __init__(self, n_states, n_actions, seed=None):
        self.n_states = n_states
        self.n_actions = n_actions

        self.random_state = np.random.RandomState(seed)

    def p(self, next_state, state, action):
        raise NotImplementedError()

    def r(self, next_state, state, action):
        raise NotImplementedError()

    def draw(self, state, action):
        p = [self.p(ns, state, action) for ns in range(self.n_states)]
        next_state = self.random_state.choice(self.n_states, p=p)
        reward = self.r(next_state, state, action)

        return next_state, reward


class Environment(EnvironmentModel):
    def __init__(self, n_states, n_actions, max_steps, pi, seed=None):
        EnvironmentModel.__init__(self, n_states, n_actions, seed)

        self.max_steps = max_steps

        self.pi = pi
        if self.pi is None:
            self.pi = np.full(n_states, 1./n_states)

    def reset(self):
        self.n_steps = 0
        self.state = self.random_state.choice(self.n_states, p=self.pi)

        return self.state

    def step(self, action):
        if action < 0 or action >= self.n_actions:
            raise Exception('Invalid action.')
```

```
        self.n_steps += 1
        done = (self.n_steps >= self.max_steps)

        self.state, reward = self.draw(self.state, action)

        return self.state, reward, done

    def render(self, policy=None, value=None):
        raise NotImplementedError()


class FrozenLake(Environment):
    def __init__(self, lake, slip, max_steps, seed=None):
    """
    lake: A matrix that represents the lake. For example:
     lake =  [['&', '.', '.', '.'],
              ['.', '#', '.', '#'],
              ['.', '.', '.', '#'],
              ['#', '.', '.', '$']]
    slip: The probability that the agent will slip
    max_steps: The maximum number of time steps in an episode
    seed: A seed to control the random number generator (optional)
    """
        # start (&), frozen (.), hole (#), goal ($)
        self.lake = np.array(lake)
        self.lake_flat = self.lake.reshape(-1)

        self.slip = slip

        n_states = self.lake.size + 1
        n_actions = 4

        pi = np.zeros(n_states, dtype=float)
        pi[np.where(self.lake_flat == '&')[0]] = 1.0

        self.absorbing_state = n_states - 1

        # TODO:

    def step(self, action):
        state, reward, done = Environment.step(self, action)

        done = (state == self.absorbing_state) or done

        return state, reward, done

    def p(self, next_state, state, action):
        # TODO:

    def r(self, next_state, state, action):
        # TODO:

    def render(self, policy=None, value=None):
        if policy is None:
            lake = np.array(self.lake_flat)

            if self.state < self.absorbing_state:
                lake[self.state] = '@'

            print(lake.reshape(self.lake.shape))
        else:
```

```
            # UTF-8 arrows look nicer, but cannot be used in LaTeX
            # https://www.w3schools.com/charsets/ref_utf_arrows.asp
            actions = ['^', '<', '_', '>']

            print('Lake:')
            print(self.lake)

            print('Policy:')
            policy = np.array([actions[a] for a in policy[:-1]])
            print(policy.reshape(self.lake.shape))

            print('Value:')
            with _printoptions(precision=3, suppress=True):
                print(value[:-1].reshape(self.lake.shape))

def play(env):
    actions = ['w', 'a', 's', 'd']

    state = env.reset()
    env.render()

    done = False
    while not done:
        c = input('\nMove: ')
        if c not in actions:
            raise Exception('Invalid action')

        state, r, done = env.step(actions.index(c))

        env.render()
        print('Reward: {0}.'.format(r))
```

The class *EnvironmentModel* represents a model of an environment. The constructor of this class receives a number of states, a number of actions, and a seed that controls the pseudorandom number generator. Its subclasses must implement two methods: p and r. The method p returns the probability of transitioning from *state* to *next_state* given *action*. The method r returns the expected reward in having transitioned from *state* to *next_state* given *action*. The method *draw* receives a pair of state and action and returns a state drawn according to p together with the corresponding expected reward. Note that states and actions are represented by integers starting at zero. We highly recommend that you follow the same convention, since this will facilitate immensely the implementation of reinforcement learning algorithms. You can use a Python dictionary (or equivalent data structure) to map (from and to) integers to a more convenient representation when necessary. Note that, in general, agents may receive rewards drawn probabilistically by an environment, which is not supported in this simplified implementation.

The class *Environment* represents an interactive environment and inherits from *EnvironmentModel*. The constructor of this class receives a number of states, a number of actions, a maximum number of steps for interaction, a probability distribution over initial states, and a seed that controls the pseudorandom number generator. Its subclasses must implement two methods: p and r, which were already explained above. This class has two new methods: *reset* and *step*. The method *reset* restarts the interaction between the agent and the environment by setting the number of time steps to zero and drawing a state according to the probability distribution over initial states. This state is stored by the class. The method *step* receives an action and returns a next state drawn according to p, the corresponding expected reward, and a flag variable. The new state is stored by the class. This method also keeps track of how many steps have been taken. Once the number of steps matches or exceeds the pre-defined maximum number of steps, the flag variable indicates that the interaction should end.

The class *FrozenLake* represents the frozen lake environment. Your task is to implement the methods p and r for this class. The constructor of this class receives a matrix that represents a lake, a probability that the agent will slip at any given time step, a maximum number of steps for interaction, and a seed that controls the pseudorandom number generator. This class overrides the method *step* to indicate that the interaction should also end when the absorbing state is reached. The method *render* is capable of rendering the state of the environment or a pair of policy and value function.

The function *play* can be used to test your implementation of the environment before you try the next tasks.

2 Tabular model-based reinforcement learning

Your next task is to implement policy evaluation, policy improvement, policy iteration, and value iteration. You may follow the interface suggested in Listing 2.

Listing 2: Tabular model-based algorithms.

```
def policy_evaluation(env, policy, gamma, theta, max_iterations):
    value = np.zeros(env.n_states, dtype=np.float)

    # TODO:

    return value

def policy_improvement(env, value, gamma):
    policy = np.zeros(env.n_states, dtype=int)

    # TODO:

    return policy

def policy_iteration(env, gamma, theta, max_iterations, policy=None):
    if policy is None:
        policy = np.zeros(env.n_states, dtype=int)
    else:
        policy = np.array(policy, dtype=int)

    # TODO:

    return policy, value

def value_iteration(env, gamma, theta, max_iterations, value=None):
    if value is None:
        value = np.zeros(env.n_states)
    else:
        value = np.array(value, dtype=np.float)

    # TODO:

    return policy, value
```

The function *policy_evaluation* receives an environment model, a deterministic policy, a discount factor, a tolerance parameter, and a maximum number of iterations. A deterministic policy may be represented by an array that contains the action prescribed for each state.

The function *policy_improvement* receives an environment model, the value function for a policy to be improved, and a discount factor.

The function *policy_iteration* receives an environment model, a discount factor, a tolerance parameter, a maximum number of iterations, and (optionally) the initial policy.

The function *value_iteration* receives an environment model, a discount factor, a tolerance parameter, a maximum number of iterations, and (optionally) the initial value function.

3 Tabular model-free reinforcement learning

Your next task is to implement Sarsa control and Q-learning control. You may follow the interface suggested in Listing 3. We recommend that you use the small frozen lake to test your implementation, since these algorithms may need many episodes to find an optimal policy for the big frozen lake.

Listing 3: Tabular model-free algorithms.

```
def sarsa(env, max_episodes, eta, gamma, epsilon, seed=None):
    random_state = np.random.RandomState(seed)

    eta = np.linspace(eta, 0, max_episodes)
```

```
    epsilon = np.linspace(epsilon, 0, max_episodes)

    q = np.zeros((env.n_states, env.n_actions))

    for i in range(max_episodes):
        s = env.reset()
        # TODO:

    policy = q.argmax(axis=1)
    value = q.max(axis=1)

    return policy, value

def q_learning(env, max_episodes, eta, gamma, epsilon, seed=None):
    random_state = np.random.RandomState(seed)

    eta = np.linspace(eta, 0, max_episodes)
    epsilon = np.linspace(epsilon, 0, max_episodes)

    q = np.zeros((env.n_states, env.n_actions))

    for i in range(max_episodes):
        s = env.reset()
        # TODO:

    policy = q.argmax(axis=1)
    value = q.max(axis=1)

    return policy, value
```

The function *sarsa* receives an environment, a maximum number of episodes, an *initial* learning rate, a discount factor, an *initial* exploration factor, and an (optional) seed that controls the pseudorandom number generator. Note that the learning rate and exploration factor decrease linearly as the number of episodes increases (for instance, *eta[i]* contains the learning rate for episode i).

The function *q_learning* receives an environment, a maximum number of episodes, an *initial* learning rate, a discount factor, an *initial* exploration factor, and an (optional) seed that controls the pseudorandom number generator. Note that the learning rate and exploration factor decrease linearly as the number of episodes increases (for instance, *eta[i]* contains the learning rate for episode i).

Important: The ϵ-greedy policy based on Q should break ties randomly between actions that maximize Q for a given state. This plays a large role in encouraging exploration.

4 Non-tabular model-free reinforcement learning

In this task, you will treat the frozen lake environment as if it required linear action-value function approximation. Your task is to implement Sarsa control and Q-learning control using linear function approximation. In the process, you will learn that tabular model-free reinforcement learning is a special case of non-tabular model-free reinforcement learning. You may follow the interface suggested in Listing 4.

Listing 4: Non-tabular model-free algorithms.

```
class LinearWrapper:
    def __init__(self, env):
        self.env = env

        self.n_actions = self.env.n_actions
        self.n_states = self.env.n_states
        self.n_features = self.n_actions * self.n_states

    def encode_state(self, s):
        features = np.zeros((self.n_actions, self.n_features))
        for a in range(self.n_actions):
```

```
            i = np.ravel_multi_index((s, a), (self.n_states, self.n_actions))
            features[a, i] = 1.0

        return features

    def decode_policy(self, theta):
        policy = np.zeros(self.env.n_states, dtype=int)
        value = np.zeros(self.env.n_states)

        for s in range(self.n_states):
            features = self.encode_state(s)
            q = features.dot(theta)

            policy[s] = np.argmax(q)
            value[s] = np.max(q)

        return policy, value

    def reset(self):
        return self.encode_state(self.env.reset())

    def step(self, action):
        state, reward, done = self.env.step(action)

        return self.encode_state(state), reward, done

    def render(self, policy=None, value=None):
        self.env.render(policy, value)

def linear_sarsa(env, max_episodes, eta, gamma, epsilon, seed=None):
    random_state = np.random.RandomState(seed)

    eta = np.linspace(eta, 0, max_episodes)
    epsilon = np.linspace(epsilon, 0, max_episodes)

    theta = np.zeros(env.n_features)

    for i in range(max_episodes):
        features = env.reset()

        q = features.dot(theta)

        # TODO:

    return theta

def linear_q_learning(env, max_episodes, eta, gamma, epsilon, seed=None):
    random_state = np.random.RandomState(seed)

    eta = np.linspace(eta, 0, max_episodes)
    epsilon = np.linspace(epsilon, 0, max_episodes)

    theta = np.zeros(env.n_features)

    for i in range(max_episodes):
        features = env.reset()

        # TODO:

    return theta
```

The class *LinearWrapper* implements a wrapper that behaves similarly to an environment that is given to its constructor. However, the methods *reset* and *step* return a *feature matrix* when they would typically return a state s. Each row a of this feature matrix contains the feature vector $\boldsymbol{\phi}(s, a)$ that represents the pair of action and state (s, a). The method *encode_state* is responsible for representing a state by such a feature matrix. More concretely, each possible pair of state and action is represented by a different vector where all elements except one are zero. Therefore, the feature matrix has $|\mathcal{S}||\mathcal{A}|$ columns. The method *decode_policy* receives a parameter vector $\boldsymbol{\theta}$ obtained by a non-tabular reinforcement learning algorithm and returns the corresponding greedy policy together with its value function estimate.

The function *linear_sarsa* receives an environment (wrapped by *LinearWrapper*), a maximum number of episodes, an *initial* learning rate, a discount factor, an *initial* exploration factor, and an (optional) seed that controls the pseudorandom number generator. Note that the learning rate and exploration factor decay linearly as the number of episodes grows (for instance, *eta[i]* contains the learning rate for episode i).

The function *linear_q_learning* receives an environment (wrapped by *LinearWrapper*), a maximum number of episodes, an *initial* learning rate, a discount factor, an *initial* exploration factor, and an (optional) seed that controls the pseudorandom number generator. Note that the learning rate and exploration factor decay linearly as the number of episodes grows (for instance, *eta[i]* contains the learning rate for episode i).

The Q-learning control algorithm for linear function approximation is presented in Algorithm 1. Note that this algorithm uses a slightly different convention for naming variables and omits some details for the sake of simplicity (such as learning rate/exploration factor decay).

Algorithm 1 Q-learning control algorithm for linear function approximation

Input: feature vector $\boldsymbol{\phi}(s, a)$ for all state-action pairs (s, a), number of episodes N, learning rate α, probability of choosing random action ϵ, discount factor γ

Output: parameter vector $\boldsymbol{\theta}$

1: $\boldsymbol{\theta} \leftarrow \mathbf{0}$
2: **for** each i in $\{1, \ldots, N\}$ **do**
3: $s \leftarrow$ initial state for episode i
4: **for** each action a: **do**
5: $Q(a) \leftarrow \sum_i \theta_i \phi(s, a)_i$
6: **end for**
7: **while** state s is not terminal **do**
8: **if** with probability $1 - \epsilon$: **then**
9: $a \leftarrow \arg\max_a Q(a)$
10: **else**
11: $a \leftarrow$ random action
12: **end if**
13: $r \leftarrow$ observed reward for action a at state s
14: $s' \leftarrow$ observed next state for action a at state s
15: $\delta \leftarrow r - Q(a)$
16: **for** each action a': **do**
17: $Q(a') \leftarrow \sum_i \theta_i \phi(s', a')_i$
18: **end for**
19: $\delta \leftarrow \delta + \gamma \max_{a'} Q(a')$ {Note: δ is the temporal difference}
20: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \delta \boldsymbol{\phi}(s, a)$
21: $s \leftarrow s'$
22: **end while**
23: **end for**

Important: The ϵ-greedy policy based on Q should break ties randomly between actions that maximize Q (Algorithm 1, Line 9). This plays a large role in encouraging exploration.

5 Main function

Your final implementation task is to write a program that uses all the algorithms that you have implemented for this assignment. Your main function should behave analogously to the function presented in Listing 5. Using the small frozen lake as a benchmark, find and render an optimal policy using policy iteration, value iteration, Sarsa control, Q-learning control, linear Sarsa control, and linear Q-learning. For grading purposes, if your main function does not call one of these algorithms, we will assume that it is not implemented correctly.

Listing 5: Main function.

```
def main():
    seed = 0

    # Small lake
    lake =   [['&', '.', '.', '.'],
              ['.', '#', '.', '#'],
              ['.', '.', '.', '#'],
              ['#', '.', '.', '$']]

    env = FrozenLake(lake, slip=0.1, max_steps=16, seed=seed)

    print('# Model-based algorithms')
    gamma = 0.9
    theta = 0.001
    max_iterations = 100

    print('')

    print('## Policy iteration')
    policy, value = policy_iteration(env, gamma, theta, max_iterations)
    env.render(policy, value)

    print('')

    print('## Value iteration')
    policy, value = value_iteration(env, gamma, theta, max_iterations)
    env.render(policy, value)

    print('')

    print('# Model-free algorithms')
    max_episodes = 2000
    eta = 0.5
    epsilon = 0.5

    print('')

    print('## Sarsa')
    policy, value = sarsa(env, max_episodes, eta, gamma, epsilon, seed=seed)
    env.render(policy, value)

    print('')

    print('## Q-learning')
    policy, value = q_learning(env, max_episodes, eta, gamma, epsilon, seed=seed)
    env.render(policy, value)

    print('')

    linear_env = LinearWrapper(env)

    print('## Linear Sarsa')

    parameters = linear_sarsa(linear_env, max_episodes, eta,
                              gamma, epsilon, seed=seed)
    policy, value = linear_env.decode_policy(parameters)
    linear_env.render(policy, value)

    print('')

    print('## Linear Q-learning')
```

```
parameters = linear_q_learning(linear_env, max_episodes, eta,
                               gamma, epsilon, seed=seed)
policy, value = linear_env.decode_policy(parameters)
linear_env.render(policy, value)
```

6 Submission instructions

This assignment corresponds to 40% of the final grade for this module. You will work in groups of 3 students. The deadline for submitting this assignment is December 11th, 2020. Penalties for late submissions will be applied in accordance with the School policy. The submission cut-off date is 7 days after the deadline. Submissions should be made through QM+. Submissions by e-mail will be ignored. Please always check whether the files were uploaded correctly to QM+. Cases of extenuating circumstances have to go through the proper procedure in accordance with the School policy. Only cases approved by the School in due time will be considered.

You will find the group selection page in QM+. You must be part of a group in QM+ before submitting your assignment. Plagiarism leads to irreversible non-negotiable failure in the module. If you are unsure about what constitutes plagiarism, please ask.

This assignment requires a group submission and an individual submission, which are detailed in the next sections.

6.1 Group submission

The group submission must be a single zip file. This file must contain a single folder named *group[group id]*. This folder must contain a report and a folder named *code*.

The *code* folder must contain a file named *README.txt*, which explains how to run your main function (see Section 5). Based solely on the correctness and clarity of your code, you will receive the following number of points for accomplishing each of the following tasks:

1. Implementing the frozen lake environment [10/100]
2. Implementing policy iteration [10/100]
3. Implementing value iteration [10/100]
4. Implementing Sarsa control [10/100]
5. Implementing Q-learning [10/100]
6. Implementing Sarsa control using linear function approximation [10/100]
7. Implementing Q-learning control using linear function approximation [10/100]

The report must be a single pdf file. Other formats are not acceptable. The report must be excellently organized and identified with your names, student numbers, and module identifier. You will receive the following number of points for answering each of the following questions:

1. Explain how your code for this assignment is organized. Did you make implementation decisions that deviate significantly from what we suggested? [10/100]
2. How many iterations did policy iteration require to find an optimal policy for the big frozen lake? How many iterations did value iteration require? Which algorithm was faster? [10/100]
3. How many episodes did Sarsa control require to find an optimal policy for the small frozen lake? How many episodes did Q-learning control require? **Hint:** you may use policy evaluation to compare the value of each policy obtained by these algorithms to the value of an optimal policy [10/100]
4. In linear action-value function approximation, how can each element of the parameter vector $\boldsymbol{\theta}$ be interpreted when each possible pair of state s and action a is represented by a different feature vector $\boldsymbol{\phi}(s, a)$ where all elements except one are zero? Explain why the tabular model-free reinforcement learning algorithms that you implemented are a special case of the non-tabular model-free reinforcement learning algorithms that you implemented. [Bonus 5/100]
5. Try to find an optimal policy for the big frozen lake by tweaking the parameters for Sarsa control and Q-learning control (maximum number of episodes, learning rate, and exploration factor). You must use policy evaluation to confirm that the resulting policy is optimal. Even if you fail, describe your experience. [Bonus 5/100]

6.2 Individual submission

Each student must submit a text that describes in no more than 300 words the role that each member of their group had in the assignment. Your grade may be penalized if the individual submissions reveal that you have not contributed enough to the assignment. The individual submission must be a single pdf file. Other formats are not acceptable.