Assignment 2

Statistical Planning and Reinforcement Learning

In the first part of this assignment, you will implement a variety of reinforcement learning algorithms to find policies for the *frozen lake* environment. In the second part of this assignment, you will experiment with environments and algorithms of your choice.

Please read this entire document before you start working on the assignment.

1 The frozen lake

1.1 Environment

The code presented in this section uses the NumPy library. If you are not familiar with NumPy, please read the NumPy quickstart tutorial and the NumPy broadcasting tutorial.

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, which may be the desired direction). An action that would cause the agent to move outside the grid leaves the state unchanged.

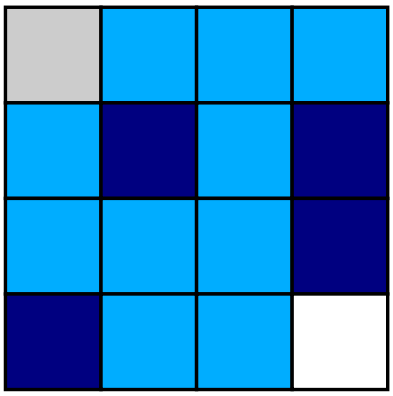

Figure 1: Small frozen lake

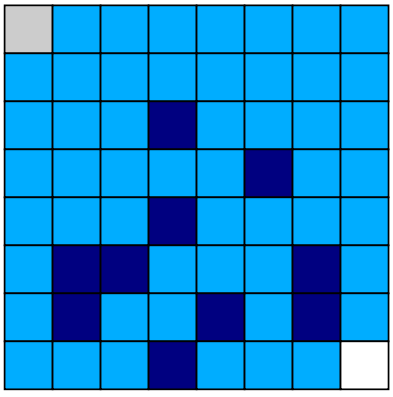

Figure 2: Big frozen lake

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 Python, try to mimic the 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:

        Environment.__init__(self, n_states, n_actions, max_steps, pi, seed=seed)

    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. Your implementation must similarly be capable of receiving any matrix that represents a lake.

The function *play* can be used to interactively test your implementation of the environment.

Important: Implementing the frozen lake environment is deceptively simple. The file *p.npy* contains a *numpy.array* with the probability to be returned by the method p for each combination of *next_state*, *state*, and *action* in the small frozen lake. You should load this file using *numpy.load* to check your implementation. The tiles are numbered in row-major order, with the absorbing state coming last. The actions are numbered in the following order: up, left, down, and right.

1.2 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.

1.3 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.

1.4 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. The a-th row 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, exploration factor 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.

1.5 Deep reinforcement learning

The code presented in this section uses the PyTorch library. If you are not familiar with PyTorch, please read the Learn the Basics tutorial.

In this task, you will implement the deep Q-network learning algorithm [Mnih et al., 2015] and treat the frozen lake environment as if it required non-linear action-value function approximation. In the process, you will learn how to train a reinforcement learning agent that receives images as inputs. You may follow the interface suggested in Listing 5.

Listing 5: Deep reinforcement learning algorithms.

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

        lake = self.env.lake

        self.n_actions = self.env.n_actions
        self.state_shape = (4, lake.shape[0], lake.shape[1])

        lake_image = [(lake == c).astype(float) for c in ['&', '#', '$']]

        self.state_image = {lake.absorbing_state:
                            np.stack([np.zeros(lake.shape)] + lake_image)}
        for state in range(lake.size):
```

```
            # TODO:

    def encode_state(self, state):
        return self.state_image[state]

    def decode_policy(self, dqn):
        states = np.array([self.encode_state(s) for s in range(self.env.n_states)])
        q = dqn(states).detach().numpy()  # torch.no_grad omitted to avoid import

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

        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)


class DeepQNetwork(torch.nn.Module):
    def __init__(self, env, learning_rate, kernel_size, conv_out_channels,
                 fc_out_features, seed):
        torch.nn.Module.__init__(self)
        torch.manual_seed(seed)

        self.conv_layer = torch.nn.Conv2d(in_channels=env.state_shape[0],
                                          out_channels=conv_out_channels,
                                          kernel_size=kernel_size, stride=1)

        h = env.state_shape[1] - kernel_size + 1
        w = env.state_shape[2] - kernel_size + 1

        self.fc_layer = torch.nn.Linear(in_features=h * w * conv_out_channels,
                                        out_features=fc_out_features)
        self.output_layer = torch.nn.Linear(in_features=fc_out_features,
                                            out_features=env.n_actions)

        self.optimizer = torch.optim.Adam(self.parameters(), lr=learning_rate)

    def forward(self, x):
        x = torch.tensor(x, dtype=torch.float)

        # TODO:

    def train_step(self, transitions, gamma, tdqn):
        states = np.array([transition[0] for transition in transitions])
        actions = np.array([transition[1] for transition in transitions])
        rewards = np.array([transition[2] for transition in transitions])
        next_states = np.array([transition[3] for transition in transitions])
        dones = np.array([transition[4] for transition in transitions])

        q = self(states)
        q = q.gather(1, torch.Tensor(actions).view(len(transitions), 1).long())
        q = q.view(len(transitions))

        with torch.no_grad():
            next_q = tdqn(next_states).max(dim=1)[0] * (1 - dones)

        target = torch.Tensor(rewards) + gamma * next_q
```

```
        # TODO: the loss is the mean squared error between `q` and `target`
        loss = 0

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()


class ReplayBuffer:
    def __init__(self, buffer_size, random_state):
        self.buffer = deque(maxlen=buffer_size)
        self.random_state = random_state

    def __len__(self):
        return len(self.buffer)

    def append(self, transition):
        self.buffer.append(transition)

    def draw(self, batch_size):
        # TODO:


def deep_q_network_learning(env, max_episodes, learning_rate, gamma, epsilon,
                            batch_size, target_update_frequency, buffer_size,
                            kernel_size, conv_out_channels, fc_out_features, seed):
    random_state = np.random.RandomState(seed)
    replay_buffer = ReplayBuffer(buffer_size, random_state)

    dqn = DeepQNetwork(env, learning_rate, kernel_size, conv_out_channels,
                       fc_out_features, seed=seed)
    tdqn = DeepQNetwork(env, learning_rate, kernel_size, conv_out_channels,
                        fc_out_features, seed=seed)

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

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

        done = False
        while not done:
            if random_state.rand() < epsilon[i]:
                action = random_state.choice(env.n_actions)
            else:
                with torch.no_grad():
                    q = dqn(np.array([state]))[0].numpy()

                qmax = max(q)
                best = [a for a in range(env.n_actions) if np.allclose(qmax, q[a])]
                action = random_state.choice(best)

            next_state, reward, done = env.step(action)

            replay_buffer.append((state, action, reward, next_state, done))

            state = next_state

            if len(replay_buffer) >= batch_size:
                transitions = replay_buffer.draw(batch_size)
                dqn.train_step(transitions, gamma, tdqn)

        if (i % target_update_frequency) == 0:
            tdqn.load_state_dict(dqn.state_dict())

    return dqn
```

The class *FrozenLakeImageWrapper* implements a wrapper that behaves similarly to a frozen lake environment that must be given to its constructor. However, the methods *reset* and *step* return an image when they would typically return a state. This image is composed of four channels and is represented by a *numpy.array* of shape $(4, h, w)$, where h is the number of rows and w is the number of columns of the lake grid. The first channel of this image is a $h \times w$ matrix whose elements are all zero except for the element that corresponds to the position of the agent, which has value one. The second channel of this image is a $h \times w$ matrix whose elements are all zero except for the element that corresponds to the start tile, which has value one. The third channel of this image is a $h \times w$ matrix whose elements are all zero except for the elements that correspond to hole tiles, which have value one. The fourth channel of this image is a $h \times w$ matrix whose elements are all zero except for the element that corresponds to the goal tile, which has value one. The method *decode_policy* receives a neural network obtained by the deep Q-network learning algorithm and returns the corresponding greedy policy together with its value function estimate.

The class *DeepQNetwork* implements a convolutional neural network used by the deep Q-network learning algorithm. The class constructor is responsible for creating each layer (and associated parameters) and the optimizer object (which uses a gradient-based optimization method called Adam [Kingma and Ba, 2015]). The method *forward* receives a list (or *numpy.array*) containing B states (images), creates an input tensor of shape $(B, 4, h, w)$, and outputs a tensor of shape $(B, |\mathcal{A}|)$. For each state in the input list, the output tensor has a row that contains the action-value estimate of each action in that state. The output tensor is obtained by transforming the input tensor by the convolutional layer; a rectified linear unit activation function; the fully connected layer; a rectified linear unit activation function; and the output layer. The method *train_step* receives a list of transitions (batch), computes a loss function (mean squared temporal differences) given this batch, computes the gradient of this loss with respect to the network parameters, and uses this gradient for one optimization step.

The class *ReplayBuffer* implements a replay buffer that stores transitions. A transition is a tuple composed of a state, action, reward, next state, and a flag variable that denotes whether the episode ended at the next state. The buffer is represented by a Python *deque* object that automatically discards the oldest transitions when it reaches capacity. The method *draw* returns a list of *batch_size* transitions drawn without replacement from the replay buffer.

Finally, the function *deep_q_network_learning* combines the previous classes to implement the deep Q-network learning algorithm.

1.6 Main function

Your final implementation task of the first part of the assignment is to write a program that uses all the algorithms that you have implemented so far. Your main function should behave analogously to the function presented in Listing 6. Using the small frozen lake as a benchmark, find and render optimal policies and values using policy iteration, value iteration, Sarsa control, Q-learning control, linear Sarsa control, linear Q-learning, and deep Q-network learning. For marking purposes, if your main function does not call one of these algorithms, we will assume that it is not implemented correctly.

Listing 6: Main function.

```
def main():
    seed = 0

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

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

    env = FrozenLake(lake, slip=0.1, max_steps=16, seed=seed)
    gamma = 0.9
```

```
print('# Model-based algorithms')

print('')

print('## Policy iteration')
policy, value = policy_iteration(env, gamma, theta=0.001, max_iterations=128)
env.render(policy, value)

print('')

print('## Value iteration')
policy, value = value_iteration(env, gamma, theta=0.001, max_iterations=128)
env.render(policy, value)

print('')

print('# Model-free algorithms')
max_episodes = 4000

print('')

print('## Sarsa')
policy, value = sarsa(env, max_episodes, eta=0.5, gamma=gamma,
                      epsilon=0.5, seed=seed)
env.render(policy, value)

print('')

print('## Q-learning')
policy, value = q_learning(env, max_episodes, eta=0.5, gamma=gamma,
                           epsilon=0.5, seed=seed)
env.render(policy, value)

print('')

linear_env = LinearWrapper(env)

print('## Linear Sarsa')

parameters = linear_sarsa(linear_env, max_episodes, eta=0.5, gamma=gamma,
                          epsilon=0.5, 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=0.5, gamma=gamma,
                               epsilon=0.5, seed=seed)
policy, value = linear_env.decode_policy(parameters)
linear_env.render(policy, value)

print('')

image_env = FrozenLakeImageWrapper(env)

print('## Deep Q-network learning')

dqn = deep_q_network_learning(image_env, max_episodes, learning_rate=0.001,
                              gamma=gamma, epsilon=0.2, batch_size=32,
                              target_update_frequency=4, buffer_size=256,
                              kernel_size=3, conv_out_channels=4,
                              fc_out_features=8, seed=4)
policy, value = image_env.decode_policy(dqn)
image_env.render(policy, value)
```

The expected output of the main function is presented in Listing 7. The model-free algorithms may fail to find an optimal policy and their value estimates may differ even after an optimal policy is found.

Listing 7: Main function output.

```
# UTF-8 arrows look nicer, but cannot be used in LaTeX
# https://www.w3schools.com/charsets/ref_utf_arrows.asp

# Model-based algorithms

## Policy iteration
Lake:
[['&' '.' '.' '.']
 ['.' '#' '.' '#']
 ['.' '.' '.' '#']
 ['#' '.' '.' '$']]
Policy:
[['_' '>' '_' '<']
 ['_' '^' '_' '^']
 ['>' '_' '_' '^']
 ['^' '>' '>' '^']]
Value:
[[0.455 0.504 0.579 0.505]
 [0.508 0.    0.653 0.   ]
 [0.584 0.672 0.768 0.   ]
 [0.    0.771 0.887 1.   ]]

## Value iteration
Lake:
[['&' '.' '.' '.']
 ['.' '#' '.' '#']
 ['.' '.' '.' '#']
 ['#' '.' '.' '$']]
Policy:
[['_' '>' '_' '<']
 ['_' '^' '_' '^']
 ['>' '_' '_' '^']
 ['^' '>' '>' '^']]
Value:
[[0.455 0.504 0.579 0.505]
 [0.508 0.    0.653 0.   ]
 [0.584 0.672 0.768 0.   ]
 [0.    0.771 0.887 1.   ]]

# Model-free algorithms

## Sarsa
Lake:
[['&' '.' '.' '.']
 ['.' '#' '.' '#']
 ['.' '.' '.' '#']
 ['#' '.' '.' '$']]
Policy:
[['_' '>' '_' '<']
 ['_' '^' '_' '^']
 ['>' '_' '_' '^']
 ['^' '>' '>' '^']]
Value:
[[0.445 0.386 0.47  0.28 ]
 [0.5   0.    0.583 0.   ]
 [0.589 0.687 0.75  0.   ]
 [0.    0.79  0.893 1.   ]]

## Q-learning
Lake:
[['&' '.' '.' '.']
 ['.' '#' '.' '#']
```

```
 ['.' '.' '.' '#']
 ['#' '.' '.' '$']]
Policy:
[['-' '<' '-' '<']
 ['-' '^' '-' '^']
 ['>' '-' '-' '^']
 ['^' '>' '>' '^']]
Value:
[[0.453 0.385 0.53  0.443]
 [0.51  0.    0.638 0.   ]
 [0.601 0.675 0.773 0.   ]
 [0.    0.758 0.884 1.   ]]
```

Linear Sarsa

```
Lake:
[['&' '.' '.' '.']
 ['.' '#' '.' '#']
 ['.' '.' '.' '#']
 ['#' '.' '.' '$']]
Policy:
[['-' '>' '-' '<']
 ['-' '^' '-' '^']
 ['>' '-' '-' '^']
 ['^' '>' '>' '^']]
Value:
[[0.43  0.434 0.517 0.371]
 [0.5   0.    0.655 0.   ]
 [0.578 0.678 0.763 0.   ]
 [0.    0.778 0.887 1.   ]]
```

Linear Q-learning

```
Lake:
[['&' '.' '.' '.']
 ['.' '#' '.' '#']
 ['.' '.' '.' '#']
 ['#' '.' '.' '$']]
Policy:
[['-' '<' '-' '>']
 ['-' '^' '-' '^']
 ['>' '>' '-' '^']
 ['^' '>' '>' '^']]
Value:
[[0.444 0.377 0.478 0.376]
 [0.505 0.    0.637 0.   ]
 [0.588 0.67  0.773 0.   ]
 [0.    0.769 0.886 1.   ]]
```

Deep Q-network learning

```
Lake:
[['&' '.' '.' '.']
 ['.' '#' '.' '#']
 ['.' '.' '.' '#']
 ['#' '.' '.' '$']]
Policy:
[['-' '>' '^' '^']
 ['-' '-' '-' '-']
 ['>' '-' '-' '<']
 ['-' '>' '>' '-']]
Value:
[[ 0.405  0.399  0.503  0.528]
 [ 0.447 -0.004  0.646  0.064]
 [ 0.486  0.567  0.741  0.002]
 [ 0.001  0.655  0.874  1.013]]
```

1.7 Questions

Your report (detailed in Section 3.2) should also answer each each of the following questions:

1. How many iterations did policy iteration require before returning an optimal policy for the big frozen lake? How many iterations did value iteration require? [1.25/15]
2. For each model-free reinforcement learning algorithm (Sarsa control, Q-learning control, linear Sarsa control, linear Q-learning control, deep Q-network learning) store the return (sum of **discounted** rewards) obtained during each episode of interaction with the small frozen lake. For each of these algorithms, include a plot that shows the episode number on the x-axis and a moving average of these values on the y-axis. Use a moving average window of length 20. *Hint: np.convolve(returns_array, np.ones(20)/20, mode='valid').* [5/15]
3. Try to minimize the number of episodes required to find an optimal policy for the *small* frozen lake by tweaking the parameters of Sarsa control and Q-learning control (learning rate and exploration factor). Describe your results. Then try to find an optimal policy for the *big* frozen lake by tweaking the parameters of Sarsa control and Q-learning control. Even if you fail, describe your results. [5/15]
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. [1.25/15]
5. During deep Q-network training, why is it necessary to act according to an ϵ-greedy policy instead of a greedy policy (with respect to Q)? [1.25/15]
6. How do the authors of deep Q-network learning [Mnih et al., 2015] explain the need for a target Q-network in addition to an online Q-network? [1.25/15]

1.8 Marks

This part of the assignment is worth 50 points. Based on the correctness and clarity of your code, the output of your main function, and the content of your report, you will receive the following number of points for each of the following tasks:

1. Implementing the frozen lake environment [5/50]
2. Implementing policy iteration [3.75/50]
3. Implementing value iteration [3.75/50]
4. Implementing Sarsa control [3.75/50]
5. Implementing Q-learning control [3.75/50]
6. Implementing Sarsa control using linear function approximation [3.75/50]
7. Implementing Q-learning control using linear function approximation [3.75/50]
8. Implementing deep Q-network learning [7.5/50]
9. Answering the questions in Section 1.7 [15/50]

2 Beyond the frozen lake

In this part of the assignment, you will experiment with environments and algorithms of your choice.

2.1 Environments

You should find or develop a highly customizable environment for your experiments. You will not be rewarded for implementing environments from scratch when viable alternatives exist. In other words, you are highly encouraged to use existing libraries or develop wrappers on top of existing libraries. For example, Minigrid is a library that allows creating highly customized grid world environments. The Farama Foundation maintains other high-quality libraries. You will be rewarded for creative or challenging choices of environments.

Your report should contain a clear description of your chosen environment(s) in terms of states, actions, rewards, and other characteristics. You should also justify your choice.

2.2 Algorithms

You should find or develop reinforcement learning algorithms for your experiments. You will not be rewarded for implementing algorithms from scratch when viable alternatives exist. In other words, you are highly encouraged to use existing libraries or develop wrappers on top of existing libraries. For example, Stable Baselines is a library that reliably implements modern reinforcement learning algorithms. You will be rewarded for creative or challenging choices of algorithms, specially if they address recognized reinforcement learning challenges [Arulkumaran et al., 2017, Section VI].

Your report should contain a brief overview of your chosen algorithm(s) and its (their) hyperparameters. You should also justify your choice.

2.3 Experiments

You should experiment with your choice of environments and algorithms using an experimental protocol that allows answering interesting (open-ended) questions. For example, if you choose to use Minigrid and Stable Baselines, you can study the sample efficiency of modern reinforcement learning algorithms. You will be rewarded for creative or challenging experiments. You will have to decide which relevant metrics (for example, return or execution time) should be recorded. You may need to repeat your experiments to ensure that your conclusions are reliable.

Your report should contain a thoroughly unambiguous description of your experimental protocol that would enable it to be reproduced. Naturally, this should include environment and algorithm settings.

2.4 Analysis

Your report should present the results of your experiments through appropriate usage of tables and plots. Your report should also provide a clear interpretation of the results. You will be rewarded for formulating and testing hypotheses, specially if they are insightful or connected with current research. You should also discuss the strengths and weaknesses of your experimental protocol and suggest ideas for additional experiments.

2.5 Marks

Because this part of the assignment is open-ended, you should talk to the module team about your ideas to receive timely feedback. **In order to encourage engagement throughout the semester, we will only provide feedback during lab sessions.**

Based on the correctness and clarity of your code and the content of your report, you will receive the following number of points for each of the following tasks:

1. Choosing creative or challenging environments/algorithms/experiments [35/50]
2. Implementing a conclusive and reproducible experimental protocol [10/50]
3. Conducting a rigorous analysis of the results [5/50]

3 Submission

This assignment requires three separate submissions, which are detailed in the next sections.

This assignment corresponds to 40% of the final mark for this module. You will work in groups of 3 students. The deadline for submitting this assignment can be found on QM+. 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. We will take action if there is evidence of plagiarism. If you are unsure about what constitutes plagiarism, please ask.

3.1 Code

The code for this assignment must be submitted as a single zip file named *group[group id].zip* that contains a single folder named *group[group id]*.

The frozen lake. The folder mentioned above must contain a file named *output.txt* that stores the output of your main function. An incorrect implementation of the frozen lake environment will compromise the points received for correct implementations of reinforcement learning algorithms, since correctness will be mostly assessed based on the output of the main function. If you are not able to implement the frozen lake environment correctly, you may use the transition probabilities from *p.npy* (see Sec. 1.1).

Beyond the frozen lake. Ensure that every dependency required by your project is easy to install and that the code enables reproducing your results.

3.2 Report

The report for this assignment must be submitted as a single pdf file named *group[group id].pdf*. Other file formats are not acceptable. The report must be excellently organized and identified with your names, student numbers, and module identifier. **The report should not have more than 5 pages (including references).**

The frozen lake. The report must explain how your code for this part is organized and justify implementation decisions that deviate significantly from what we suggested. It should also contain the answer to your questions.

Beyond the frozen lake. The section of the report that covers this part of the assignment should have the same subsections as Section 2 (except for Marks) and its content should be organized accordingly.

3.3 Reflection

Individually, each student must submit a single pdf file named *[student name].pdf* that briefly describes the role that each member of the group had in the submission. Your mark may be penalized if these reflections reveal that you have not contributed enough to the submission.

References

[Arulkumaran et al., 2017] Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*.

[Kingma and Ba, 2015] Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *ICLR*.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.