# Project 3: *MDP and Reinforcement Learning*

Due Date: April 9, 11:59pm (PST)

## 1 Problem Statement

Walking on the frozen lake can be the fun thing that people living in Southern California always desire to try once in a lifetime. Though it is hard to have such an experience in the real life, you can try as many times as you want in this project. However, walking on the frozen lake is dangerous since the lake is not completely frozen. If you make the wrong step, you may fall into the hole in the ice.

The lake you will walk on is a square lake in the shape of a grid as shown in Fig. 1. Each spot has one of four states:

1. S: Starting point
2. G: Goal point
3. F: Frozen spot, where it is safe to walk
4. H: Hole in the ice, where it is not safe to walk

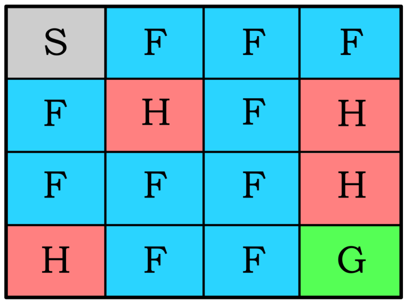


Figure 1: The lake you will walk on

The starting and goal points are always safe. We guarantee there is at least one valid path from the starting point to the goal point.

Also, you should follow the following state and action IDs to complete your implementation.

State ID:

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

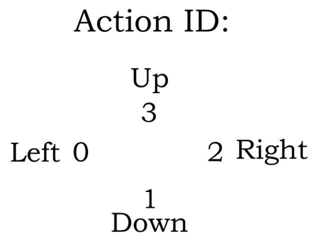


Figure 2: More configurations about walking on the frozen lake

It may be easy for you to find the safe path to walk from the starting point to the goal point, but we want you to train an agent to discover the path. However, walking on ice is not easy! Sometimes you slip and may not land on the spot you intended.

## 2 Set-up

We will use OpenAI Gym package, one of the most popular platforms for Reinforcement Learning experiments these days, for this project. You will develop your solution in Python, which is also the only support language with Gym. You can find set-up instructions for Gym here. You will use the APIs of this package, so please read it carefully. After the setup, you should successfully run the provided `code_template.py` and see the prints "Environment was loaded!" .

```
python code_template.py
```

Gym package is a collection of test problems, also called "environments". We will use the FrozenLake environment first, and the CartPole environment later.

If you are new to OpenAI Gym, it is crucial to set up the environment following the instructions above as early as possible.

Before proceeding, you can also visit the Appendix for some tools. `Numpy` library is extensively used in this project.

Permitted non-standard libraries: `gym, Numpy, Scipy`

Load the OpenAI Gym's FrozenLake environment:

```
env = gym.make('FrozenLake-v0')
```

Key configurations in `env`:

1. `env.nS`: number of states
2. `env.nA`: number of actions
3. `env.P`: environment dynamics. For example,

   ```
   current_state = 10  # State 10 from the total S_n=16 State space
   action = 0  # Left action from A_n=4 Action space
   print(env.P[current_state][action])
   ```

   Output:

   ```
   [(0.3333333333333333, 6, 0.0, False),
   (0.3333333333333333, 9, 0.0, False),
   (0.3333333333333333, 14, 0.0, False)]
   ```

   It is in the format as following:

   ```
   env.P[current_state][action] = [(prob, next_state, reward, is_terminal)]
   ```

   Given $S_t = 10, A_t = 0$ in a stochastic environment, the transition probability functions indicate that you can end up in spot 6, 9, 14, each with $1/3$ probability:

   $$\mathbb{P}[S_{t+1} = 6 | S_t = 10, A_t = 0] = \frac{1}{3}$$
   $$\mathbb{P}[S_{t+1} = 9 | S_t = 10, A_t = 0] = \frac{1}{3}$$
   $$\mathbb{P}[S_{t+1} = 14 | S_t = 10, A_t = 0] = \frac{1}{3}$$

   The reward is 0 for all the cases. All the cases are not terminated, which means you do not reach the goal and do not fall into the hole.

4. To show the demo of visualization of the lake and action executions, use

   ```
   python code_template.py -vis
   ```

   It may be helpful to check out that part of code in the template:

```
    if args.visualization:  [...].
```

For more APIs about FrozenLake environment, please refer to `code_template.py` and links above.

## 3 Task 1: Value Iteration (3 pts)

Translating pseudocode to code is critical for the reproducibility of the research. Following the pseudocode in Fig. 3, you are asked to implement value iteration to find the policy.

**Note:** initialize $V(s) = 0$ for all $s \in S$, not arbitrarily. This has been done in the code template, so do not change.

```
Value iteration

Initialize array $V$ arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
```

Figure 3: Pseudocode of value iteration

The deterministic policy means that you always go to the direction which gives the optimal future reward. If two or more directions tie, then split the probability evenly. At the holes or goal, use the policy that each action is equally likely.

To test your implementation for task 1, run the command below in the work path:

```
python code_template.py -q 1
```

For the given example in the code template, the expected output is in the description under `value_iteration` function.

## 4 Task 2: Policy Evaluation (3 pts)

Please implement policy evaluation following the pseudocode in Fig. 4 to find the state values of the given policy below. Fill out `policy_evaluation` function in `code_template.py`.

```
Policy = all actions equally likely at all the states
```

**Iterative policy evaluation**

```
Input $\pi$, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
      $v \leftarrow V(s)$
      $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$
```

Figure 4: Pseudocode of policy evaluation

To test your implementation for task 2, run the command below in the work path:

```
python code_template.py -q 2
```

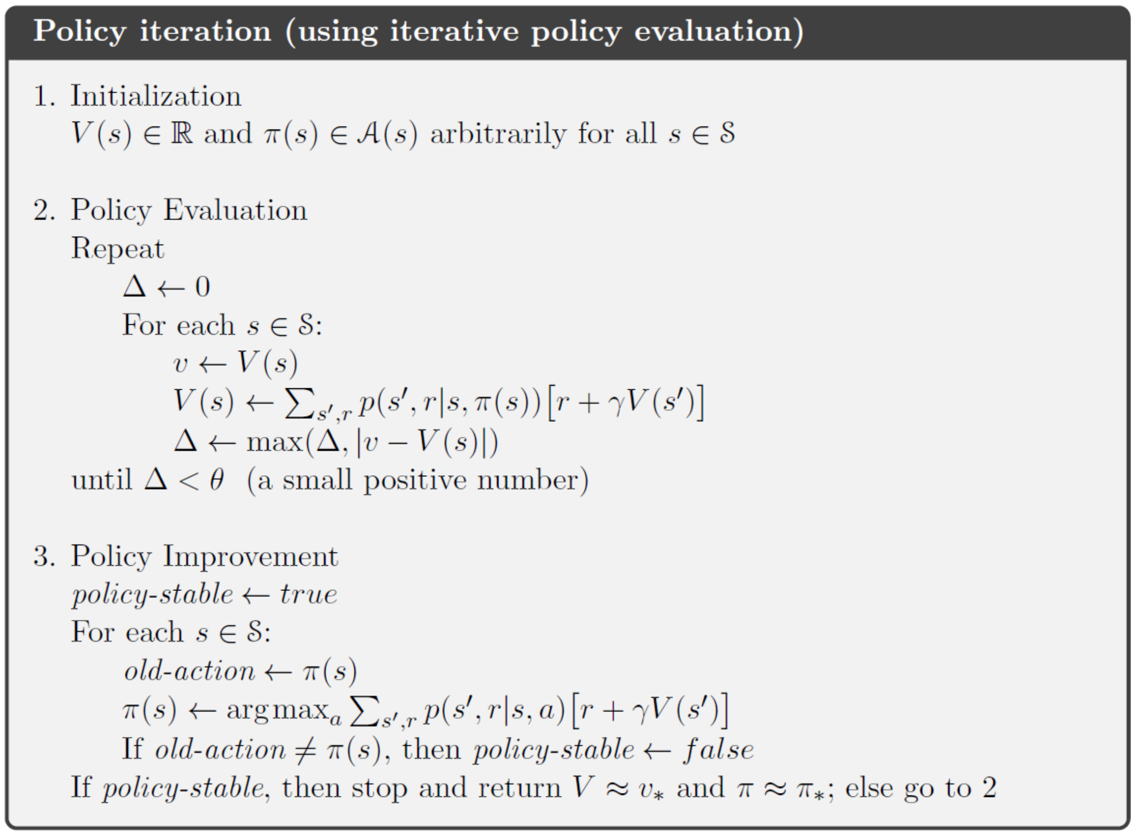
For the given example in the code template, the expected output is in the description under `policy_evalution` function.

## 5 Task 3: Policy Iteration (3 pts)

Please implement policy iteration following the pseudocode in Fig. 5 to find the policy. You will use the policy evaluation function from task 2 in this implementation. So you need to make sure your task 2 is correct before proceeding.

Fill out `policy_iteration` function in `code_template.py`.

**Note:** initialize $V(s) = 0$ and policy that each action is equally likely. This has been done in the `policy_iteration` function, so do not change.



**Policy iteration (using iterative policy evaluation)**

```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
       $\Delta \leftarrow 0$
       For each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   $\textit{policy-stable} \leftarrow \textit{true}$
   For each $s \in \mathcal{S}$:
       $\textit{old-action} \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
       If $\textit{old-action} \neq \pi(s)$, then $\textit{policy-stable} \leftarrow \textit{false}$
   If $\textit{policy-stable}$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

Figure 5: Pseudocode of policy iteration

To test your implementation for task 3, run the command below:

```
python code_template.py -q 3
```

For the given example in the code template, the expected output is in the description under `policy_iteration` function.

## 6 Task 4: Q-Learning (11 pts)

### 6.1 Implementation (9 pts)

Now let us say you do not know the environment dynamics, which means that you do not know the transition probability $\mathbb{P}$ (`env.P`). First, we implement the $\epsilon$-greedy Q-learning to learn the Q-table, and then extract policy based on the learned Q-table.

Please follow the pseudocode in Fig. 6 about the general $\epsilon$-greedy Q-learning. There are many variants of $\epsilon$-greedy Q-learning and we will implement three versions of them. They are very similar and the only difference is the action selection based on $\epsilon$-greedy policy, which corresponds to the following line in the pseudocode.

```
Choose $A$ from $S$ using policy derived from $Q$, e.g. $\epsilon$-greedy
```

If you implement version 1 correctly, you only need to make slight modification to realize version 2 and 3.

```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Figure 6: Pseudocode of Q-Learning

1. **Version 1: Vanilla $\epsilon$-greedy (3 pts)**

   When the random sampling $p$ is smaller than the threshold $\epsilon$, the agent explore the environment. Otherwise, the agent exploit the current knowledge.

   Python pseudocode:

   ```
   p = np.random.rand() # do not change, for reproducibility
   if p < epsilon:
       take random action # fill out your code
   else:
       take current best action  # fill out your code
   ```

   Implement version 1 in `q_learning_1` function in the code template.

2. **Version 2: Decay $\epsilon$-greedy (3 pts)**

   Instead of setting a constant $\epsilon$, we set an exponentially decayed $\epsilon$ as following:

   $$\epsilon(t) = \epsilon_0 \times e^{-\lambda t}$$

   where $\epsilon_0$ is the initial threshold, $\lambda$ is a constant decay rate and $t$ is the current episode.

   In addition, to prevent $\epsilon$ from vanishing ($\epsilon$ goes to 0), we set $\epsilon$ as a constant $\epsilon_{min}$ when it drops below $\epsilon_{min}$.

   Implement version 2 in `q_learning_2` function in the code template.

3. **Version 3: Effective exploiting $\epsilon$-greedy (3 pts)**

   To minimize the exploiting of useless knowledge, we only exploit knowledge (=take current best action) when the state has contained information AND the random sampling is above the threshold $\epsilon$. **It means that we only take the best action at a state when the Q values of that state are not all zeros, and the sampling is above the threshold.**

   Implement version 3 in `q_learning_3` function in the code template.

**Note:** There are a few points below which you **must** follow:

1. Instead of initializing $Q(s,a)$ arbitrarily, we initialize $Q(s,a) = 0, \forall s \in S, a \in A(s)$. It has been done in `q_learning_{1,2,3}` functions, so do not change.
2. For random sampling in $\epsilon$-greedy, use `np.random` as followings:

```
np.random.rand()  # sample a random number between 0 and 1
np.random.choice(10) # sample a random integer between 0 and 9
```

3. Initialize $S$ using `s = env.reset()`.

In the appendix, there is a more detailed pseudocode for Q-learning and you may find it helpful for your implementation.

To test your implementation for each version of Q-learning, run the command below:

```
python code_template.py -q 4-1
python code_template.py -q 4-2
python code_template.py -q 4-3
```

For the given example in the template, the expected outputs are in the description under the corresponding function.

### 6.2 Observation and Report (2 pt)

For all three versions of $\epsilon$-greedy q-learning, play with the parameters and **report the parameters which gives the policy closest to that from value iteration, and the number of states having different policies compared to value iteration**. You may want to use the `compare_policies` function provided in the template. Parameters you can tune include all the arguements of the function, except `env`, `gamma` and `num_episode`. Also, **briefly explain your observation**.

## 7 Autograder

To test your implementation from Task 1 to Task 4, run the command below:

```
python autograder.py
```

Please note this is not your final grade for Task 1-4. Your implementation will be tested by different but similar test cases after your submission.

## 8 Task 5: Validation (3 pts)

It is time to have some fun with your Q-learning implementation. We want to see if your implementation can work on some more challenging tasks, so we provide the `CartPole` testing environment in the `validate.py`. Please learn the problem from here or here.

Please fill out `q_learning_cart` function in `code_template.py`. You can use any version of the $\epsilon$-greedy q-learning from Task 4, or a combination of two versions, or a new version which is not included in Task 4. The only thing you need to modify is the dimension of the state space and action space.

**Note:**

1. Use the `get_discrete_state` function to convert the continuous state to discrete state.
2. Q-table has been initialized with the correct dimension and size. Do not change it.
3. Q-learning can be slow for such a toy problem. You may need to run many episodes to get a decent policy.

To test your implementation of q-learning for `CartPole` environment, first train your agent by using the command below:

```
python validate.py -train
```

It will generate a file named `q_table_cartpole.npy` for the final Q-table. This .npy file will be used for the test on our end. To run the simulation again with your saved Q-table, run the following command:

```
python validate.py -test
```

A demo of the learned policy can be found here. Try to keep your inverted pendulum standing as long as possible!

The grading rule will be:

| **# of alive steps** | **Grade** |
|---|---|
| $s < 50$ | 0/3 |
| $50 \leq s < 200$ | 1/3 |
| $200 \leq s < 500$ | 2/3 |
| $500 \leq s$ | 3/3 |

Lastly and most importantly, we hope this project can be a starting point, and does not discourage you from exploring RL in your future study.

## 9 Submission (1 pt)

We will test your implementation of functions `value_iteration`, `policy_evaluation`, `policy_iteration`, `q_learning_{1,2,3,cart}`, so do not change the function names.

Files you need to submit:

1. `code_template.py` for Task 1-5
2. `q_table_cartpole.npy` from Task 5
3. a pdf file for the write-up in Section 6.2. Name it with your full name.

Zip the files above and name it as `[USCID]_[FirstName]_[LastName].zip`, for example, `1981080800_Roger_Federer.zip`. Submit the zip file to Blackboard.

# Appendices

The appendices include tools and functions you may find helpful. You do not have to use all of them.

## A Python

1. Automatic unpacking

```python
x = [[0, 1], [1, 0], [0, 0], [1, 1]]
for first, second in x:
    print(first, second)
```

## B Numpy

1. np.zeros: N-dimensional tensor initialized to all 0s.
2. np.ones: N-dimensional tensor initialized to all 1s.
3. np.eye: N-dimensional tensor initialized to a diagonal matrix of 1s.
4. np.random.choice: Randomly sample from a list, allowing you to specify weights.
5. np.argmax: Index of the maximum element.
6. np.abs: Absolute value.
7. np.sum: Sum across dimensions.
8. np.exp: Calculate the exponential value

## C Detailed Q-learning pseudocode

```
Algorithm 1: Epsilon-Greedy Q-Learning Algorithm
Data: $\alpha$: learning rate, $\gamma$: discount factor, $\epsilon$: a small number
Result: A Q-table containing Q(S,A) pairs defining estimated
        optimal policy $\pi^*$
/* Initialization                                              */
Initialize Q(s,a) arbitrarily, except Q(terminal,.);
Q(terminal,.) $\leftarrow$ 0;
/* For each step in each episode, we calculate the
   Q-value and update the Q-table                              */
for each episode do
    /* Initialize state S, usually by resetting the
       environment                                             */
    Initialize state S;
    for each step in episode do
        do
            /* Choose action A from S using epsilon-greedy
               policy derived from Q                           */
            A $\leftarrow$ SELECT-ACTION(Q, S, $\epsilon$);
            Take action A, then observe reward R and next state S';
            Q(S, A) $\leftarrow$ Q(S, A) + $\alpha$ [ R + $\gamma \max_a$ Q(S', a) - Q(S, A)];
            S $\leftarrow$ S';
        while S is not terminal;
    end
end
```

Figure 7: Detailed pseudocode of Q-Learning