**CS405 Machine Learning**

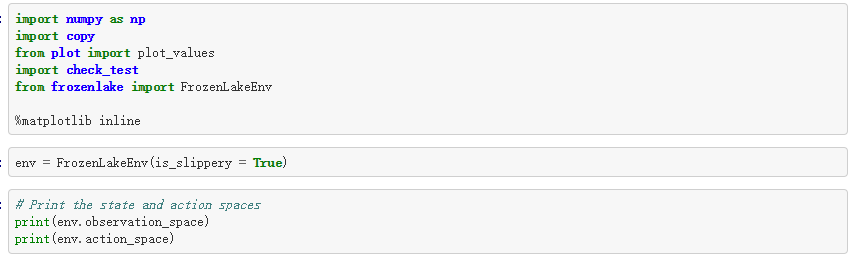
**Lab # Reinforcement Learning**

**Lab** :

Unlike policy gradient methods, which attempt to learn functions which directly map an observation to an action, Q-Learning attempts to learn the value of being in a given state, and taking a specific action there. While both approaches ultimately allow us to take intelligent actions given a situation, the means of getting to that action differ significantly.  For this Pre Lab we are going to be attempting to solve the [FrozenLake](https://gym.openai.com/envs/FrozenLake-v0" \t "_blank) environment from the [OpenAI gym](https://gym.openai.com/" \t "_blank). For those unfamiliar, the OpenAI gym provides an easy way for people to experiment with their learning agents in an array of provided toy games. The FrozenLake environment consists of a 4x4 grid of blocks, each one either being the start block, the goal block, a safe frozen block, or a dangerous hole. The objective is to have an agent learn to navigate from the start to the goal without moving onto a hole. At any given time the agent can choose to move either up, down, left, or right. The catch is that there is a wind which occasionally blows the agent onto a space they didn’t choose. As such, perfect performance every time is impossible, but learning to avoid the holes and reach the goal are certainly still doable. The reward at every step is 0, except for entering the goal, which provides a reward of 1. Thus, we will need an algorithm that learns long-term expected rewards.

**Instruction:**

Explore the environment



You will find that the environment has 16 states and 4 actions. The agent moves through a 4X4 gridworld, with states numbered as follows:

[[0 1 2 3]

[4 5 6 7]

[8 9 10 11]

[12 13 14 15]] and the agent has 4 potential actions: LEFT =0, DOWN =1, RIGHT = 2 and UP =3

Executing env.P[1][0] returns the probability of each possible reward and next state, if the agent is in state 1 of the gridworld and decides to go LEFT.



Each entry takes the form

*prob, next\_state, reward, done*

Where

* *prob* details the conditional probability of the corresponding (next\_state, reward) pair, and
* *done* is True if the *next\_state* is a terminal state, and otherwise False.

Markov Decision Process (MDP)

****state**(s)**:**** Every possible position the reinforcement learning agent could be in at any given time.

****Action**(a)**:**** What an agent can do in an environment or agent's interaction with the environment.

****Reward**(r)**:**** What the agent gets for either performing an action R(a). It could also be represented as performing an action a in a given state s denoted by R(s,a) or performing an action in a state and ending up in another state s´ denoted by R(s´|s,a)

****Transition function**T(s,a,s´)**or**P(s´|s,a)**:**** The probability that the agent is in a state s, performs an action a and ends up in another state s´ (which could be the same state i.e s≈s´)

****Policy (π):**** The is the solution to the MDP i.e. it is a function that takes in a state & returns an action π(s)->a

****The Bellmans Equation:**** Q(s,a) = r + \gamma max\_a(Q(s', a'))

Exercise 1:Model-Based Reinforcement Learning

Navigating the OpenAi’s FrozenLake environment using two model-based reinforcement learning techniques:

Value Iteration

choose initial estimate of optimal value function

repeat until change in values is sufficently small {

for each state {

calculate the maximum expected value of

neighboring state for each possible action

use maximal value of the list to update estimate of optimal value function

} each state

} convergence

calculate optimal value frunction from Bellmans' Equation

* Value iteration computes the optimal state value function by iteratively improving the estimate of V(s)
* The algorithm initializes V(s) to arbitrary random values.
* It repeatedly updates Q(s,a) & V(s) values until they converge.
* Value iteration is guaranteed to converge to optimal values.

Policy Iteration

Pseudocode:

choose initial policy & value function

repeat until policy is stable {

1. Policy evaluation:

repeat until change in values is sufficiently small {

for each state {

calculate the value of neighboring states

when taking actions according to current policy.

update estimate of optimal value function.

} each state

} converge

2. Policy improvement:

New policy according to Bellmans Equation,

assuming V^\* ≈ current V^π

} policy

* Policy iteration instead of repeated improving of the value-function estimate, it will re-define the policy at each step and compute the value according to this new policy until the policy converges.
* Policy iteration is also guaranteed to converge to the optimal policy and it often takes less iterations to converge that the value-iteration algorithm.

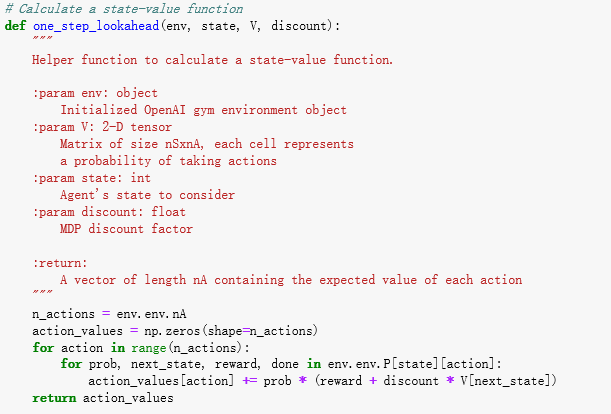
In ****Policy iteration**** algorithms, you start with a random policy, then find the value function of that policy (policy evaluation step), then find a new (improved) policy based on the previous value function, and so on. In this process, each policy is guaranteed to be a strict improvement over the previous one (unless it is already the optimal).

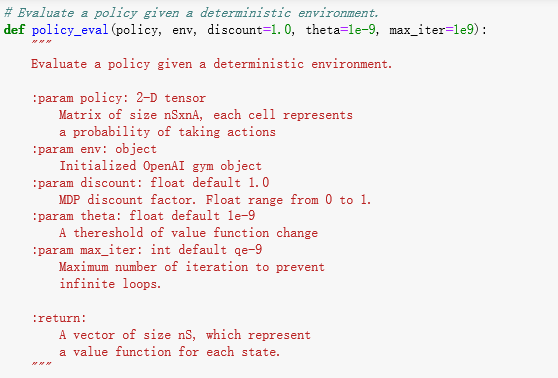
In ****Value iteration**** algorithms, you start with a random value function and then find a new (improved) value function in a iterative process, until reaching the optimal value function. Notice that you can derive easily the optimal policy from the value function by a simple argmax operation.

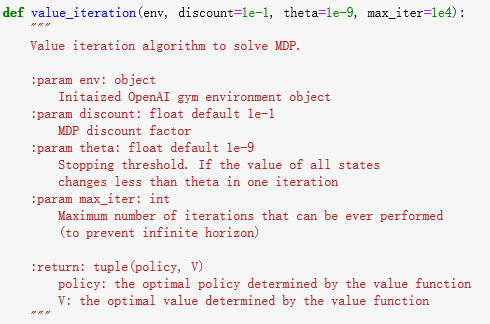
Policy iteration is generally faster than value iteration as policy converges more quickly than value function.

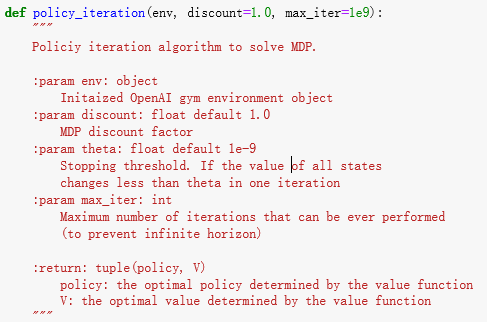
Please use policy iteration and value iteration to solve MDP respectively and output the final policy.

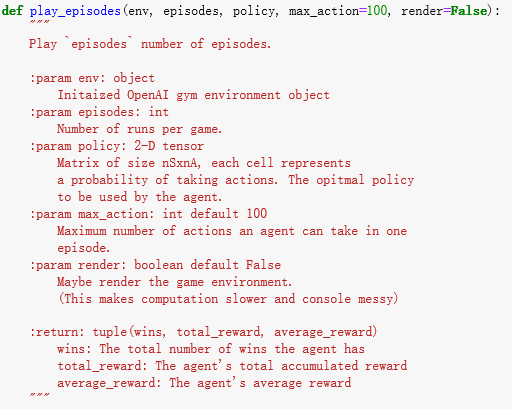
Helpers: there are lists of valuable function provided. You can infer to these functions and write your own.











Exercise 2: In it’s simplest implementation, Q-Learning is a table of values for every state (row) and action (column) possible in the environment. Within each cell of the table, we learn a value for how good it is to take a given action within a given state. In the case of the FrozenLake environment, we have 16 possible states (one for each block), and 4 possible actions (the four directions of movement), giving us a 16x4 table of Q-values. We start by initializing the table to be uniform (all zeros), and then as we observe the rewards we obtain for various actions, we update the table accordingly. We make updates to our Q-table using something called the [Bellman equation](https://en.wikipedia.org/wiki/Bellman_equation" \t "_blank), which states that the expected long-term reward for a given action is equal to the immediate reward from the current action combined with the expected reward from the best future action taken at the following state. In this way, we reuse our own Q-table when estimating how to update our table for future actions! In equation form, the rule looks like this:

*Eq 1. Q(s,a) = r + γ(max(Q(s’,a’))*

This says that the Q-value for a given state (s) and action (a) should represent the current reward (r) plus the maximum discounted (γ) future reward expected according to our own table for the next state (s’) we would end up in. The discount variable allows us to decide how important the possible future rewards are compared to the present reward. By updating in this way, the table slowly begins to obtain accurate measures of the expected future reward for a given action in a given state. Please Implement Q-table learning algorithm to solve the problem. And output the final Q-Table value.