Programming Assignment 5: Value Iteration

Strong Recommendation: Use Google CoLab for running your code, to handle the various package installs needed for this assignment.

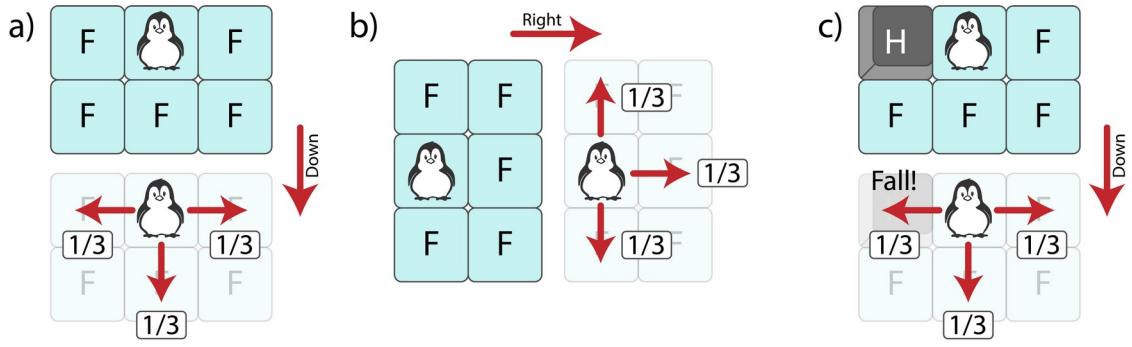
Problem Setting

Gridworld: In this assignment, we use the popular RL toolkit OpenAI Gym to implement *value iteration* on a gridworld environment (Lecture 21). In the gridworld environment we consider, an agent needs to navigate on a 2-D plane by using 4 actions (left, right, up, and down). It starts at a designated state (starting point "S"), and needs to try and reach a goal state ("G"). The agent controls the movement of a character in a grid world. Some tiles on the grid are walkable ("S", "G", and "F"), and others ("H") lead to the agent falling into the water, which ends the episode. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

Rewards: The environment is such that reaching the goal state gives the agent a `+1` reward, and it gets a `0` reward in other states (detailed description below). Hence, reaching the goal state is equivalent to maximizing the rewards it can accumulate.

Uncertainty: The environment also has some uncertainty. That is, the movement direction of the agent is uncertain and only partially depends on the chosen direction. In our setting (detailed below), each tile in the grid is "slippery", which corresponds to the following uncertainty; if the agent chose to move up/down, it will be able to do so with probability `1/3`, but with probability `2/3` it will move right/left (`1/3` right, `1/3` left). Similarly for the case that the agent chose to move right/left, it might move up/down instead. However, you will always move by 1 tile only (i.e., no "jumps").

This is visualized below:



In a) the agent (penguin) tries to walk down, in b) it tries to walk right, and c) shows a potential pitfall where just by walking by a hole "H" (and not directly into it), the agent still has a `1/3` chance to fall into the hole, which ends the episode with no reward.

Note that:

- The specific transition probabilities don't matter much here, but it is important to keep in mind is that the action you choose to move towards only partially determined where you actually move, due to the uncertianty ("slipperiness") of the environment.
- Recall that while you can see the whole gridworld (all the states), the agent only has information about its current state!

Environment: Frozen Lake

Winter is here. You and your friends are tossing around a frisbee at the park when you make a wild throw that leaves the frisbee stranded out in the middle of a lake. The water is mostly frozen ("F"), but there are a few holes ("H") where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend, and sometimes slip into another, adjacent tile! The surface is described using a grid like the following 8×8 example:

```
SFFFFFFF
FFFFFFFF
FFFHFFFF
FFFFFHFF
FFFHFFFF
FHHFFFHF
FHFFHFHF
FFFHFFFG


S : starting point, safe
F : frozen surface, safe
H : hole, fall to your doom
G : goal, where the frisbee is located
```

The episode ends when you reach the goal or fall in a hole "H". You receive a reward of 1 if you reach the goal, and 0 otherwise. The steps that you can make are one of the following: LEFT = 0, DOWN = 1, RIGHT = 2, UP = 3.

Visualization

It is easier to understand how this environment behaves if you see it first.

HINT: We **strongly recommend** that before implementing anything, you start by first running all cells of this notebook, and scroll down to the cell under **ACT7** see how a **totally-random** agent moves around the frozen lake.

You will see it fail many times. In what follows, we will implement a much better agent that can navigate to the goal.

Notation

Mapping of the OpenAI gym constructs to the notation used in lectures:

- The value function $V(s)$, is implemented here as a small array, of size corrsponsing to number of states, such that `V[s]` contains the value $V(s)$.
- The transition probabilities of the environment $P(s' \vee s, a)$ correspond to the gym object `env.P` described below.
- In lectures, the reward function $r(a \vee s, s')$ signified taking action a while in state s moving to state s'. However, in our setting here the reward fixed into the environment is such that $r(a \vee s = \text{'G'}, s') = 1$ and $r(a \vee s \neq \text{'G'}, s') = 0$ for any a, s'. In other words, in this setting we only get the reward if we *actually reach* the goal state.

First, install dependencies (takes ~1 min).

```
#remove " > /dev/null 2>&1" to see what is going on under the hood
!pip install gym pyvirtualdisplay pygame > /dev/null 2>&1
!apt-get install -y xvfb libav-tools python-opengl ffmpeg > /dev/null
2>&1
```

```
import gym
import numpy as np
import random
from IPython.display import clear_output
import time
import pygame
import matplotlib.pyplot as plt
%matplotlib inline
from IPython import display
```

```
# setting up dummy display driver for Colab
import os
os.environ['SDL_VIDEODRIVER']='dummy'
pygame.display.set_mode((640,480))
```

```
<Surface(640x480x32 SW)>
```

```
# first, we set some useful parameters:
gamma = 0.99        # discount factor
theta = 0.000001    # precision of value_iteration

# let's set up the frozen lake environment.
env = gym.make("FrozenLake8x8-v1", new_step_api=True) # create the
environment
env = env.unwrapped # enwrap it to have additional information from it

# spaces dimension
num_actions = env.action_space.n # we can move up/down/left/right,
i.e., 4 actions.
num_states = env.observation_space.n # we have one state per tile in
```

```
the grid (64 states)

# NOTE, the transition probabilities we mentioned above (i.e., the
"slipperiness"), are already given
# in the env.P object. Specifically, env.P are the transition
probabilities (dictionary dict of dicts of lists)
#           P[s][a] == [(probability, s', reward, done),
#                       (probability, s', reward, done),
#                       (probability, s', reward, done), ]

# As an example, for state s=0 (the starting tile "S"),and action a=0
(LEFT), you will have:
#           P[0][0] == [(1/3, 0, 0, False), (1/3, 0, 0, False), (1/3,
4, 0, False)]
# since:
# with probability 1/3 you'll move UP (but there's no up, so you'll
stay put at s=0),
# with probability 1/3 you'll move LEFT (but there's no left, so
you'll stay put at s=0),
# with probability 1/3 you'll move DOWN (so you'll get to the tile
below, which is s'=4).
```

Value-Iteration algorithm

We will now implement the Value Iteration algorithm you have seen in class. The pseudo-code is given below.

Pseudo-code

1. Parameter: a small threshold $\theta > 0$ determining accuracy of estimation.
2. Initialize $V(s) = -\frac{M}{1-\gamma}$, for all $s \in S$, where M is the upper bound on the absolute value of immediate rewards.
3. **Loop**:
   a. $\Delta \leftarrow 0$.
   b. **Loop for each** $s \in S$:
      i. $v \leftarrow V(s)$.
      ii. $V(s) \leftarrow \max_a \sum_{s'} p(s' \mid s,a)[r(a \mid s,s') + \gamma V(s')]$.
      iii. $\Delta \leftarrow \max$ ¿.
4. **until** $\Delta < \theta$.
5. Output a deterministic policy, $\pi \approx \pi_{¿}$, such that

$$\pi(s) = \text{argmax}_a \sum_{s'} p(s' \mid s,a)[r(a \mid s,s') + \gamma V(s')].$$

ACTS 1-2: Computing the sum & argmax of Value-Iteration.

Compute the inner sum

$$\sum_{s'} p(s' \mid s,a)[r(a \mid s,s') + \gamma V(s')],$$

given V, s, a, γ.

```
# ACT 1: compute the inner sum of the equations above, given:
        # Value function V (an array of size num_states),
        # State s, Action a, and discount factor gamma.

def sum_over_states(V, s, a, gamma):
    sum_val = 0
    # recall that env.P is a list of tuples for (s,a): [(prob, s', r,
done),...] (See block above for more details!)
    # where 'prob' is the transition probability P(s'|s,a).
    # also, since r is associated with the states, you can think of
the sum as only traversing states s'.
    # therefore, all is needed is to iterate over each possible
transition to state s'
    # from the pair (s,a), and compute the sum value (as in the
pseudo-code).
    ### YOUR CODE GOES HERE
    for state in env.P[s][a]:
      prob = state[0]
      sprime = state[1]
      r = state[2]
      sum_val += prob * (r + (gamma * V[sprime]))
    return sum_val
```

We now compute the argmax operation, in which we will update $\pi(a \vee s)$ where s is the given state, and a is the maximizing action! To obtain it, you should use the function above `sum_over_states`.

```
# ACT 2: update the action which can take us to a higher valued state,
given:
        # Value function V (an array of size num_states),
        # Policy array pi (a matrix of shape [num_states,
num_actions]),
        # State s, and discount factor gamma.

def argmax_over_actions(V, pi, s, gamma):
    max_val = 0
    ### YOUR CODE GOES HERE
    max_action = 0
    for a in range(len(pi[s])):
      val = sum_over_states(V, s, a, gamma)
      if (val > max_val):
        max_val = val
        max_action = a
    pi[s][:] = 0
    pi[s][max_action] = 1
```

```
    return pi, max_val
```

ACT 3: Computing the Bellman update

We now perform a single iteration of update to the value function V, given state s, and discount factor γ. You may call a function you have implemented above, and output the difference between the new, updated value for this state, and the previous value (denoted as `delta` in the pseudo-code above).

```
# ACT 3: update the Value function for state s, that is: V[s], by
taking action which maximizes current value. Given:
        # Value function V (an array of size num_states),
        # State s, and discount factor gamma.

def bellman_optimality_update(V, s, gamma):
    pi = np.zeros((num_states, num_actions))
    ### YOUR CODE GOES HERE
        # ACT3a: call a function implemented above, to get the action
which maximizes current value.
    pi, max_val = argmax_over_actions(V, pi, s, gamma)
        # ACT3b: update value V[s]
    old_val = V[s]
    V[s] = max_val
        # ACT 3c: compute and output delta.
    delta = abs(old_val - V[s])
    return delta
```

ACT 4: Initialize V

Initialize $V = -\frac{M}{1-\gamma}$, where M is the upper bound on the absolute value of immediate rewards to be filled in.

```
# ACT 4: Initialize V
# length_V gives the size of the vector V
def init_V(gamma, M, length_V):
    ### YOUR CODE GOES HERE
    V = np.zeros(length_V)
    V.fill((-1) * M / (1 - gamma))
    return V
```

ACT 5-6: Value-iteration (putting it all together)

```
# Now, let's put it all together. Recall that we are given:
        # discount factor gamma, and wanted precision theta.
def value_iteration(gamma, theta):
    # Initialize V with init_V function
    ### YOUR CODE GOES HERE
    V = init_V(gamma, 1, num_states)
    # ACT 5: construct the main loop,
```

```
    #         iteratively calling the bellman update,
    #         and break when sufficient accuracy was reached.
    while True:
        delta = 0
        ### YOUR CODE GOES HERE
        for s in range(len(V)):
          v = V[s]
          delta = max(bellman_optimality_update(V, s, gamma), delta)
        if (delta < theta):
          break

    pi = np.zeros((num_states, num_actions))
    # ACT 6: Extract optimal policy
    ### YOUR CODE GOES HERE
    for s in range(len(V)):
        pi, max_val = argmax_over_actions(V, pi, s, gamma)
    # output optimal value funtion, optimal policy
    return V, pi
```

Helper functions (no action needed)

This next two functions `show_state` and `pretty_print` are helper functions we provide that print and visualize state information.

```
def show_state(env, trial=0, step=0):
    plt.figure(5)
    plt.clf()
    plt.imshow(env.render(mode='rgb_array'))
    plt.title(f"Trial: {trial} | Step: {step}")
    plt.axis('off')
    clear_output(wait=True)
    display.display(plt.gcf())

def pretty_print(i_episode, t, done, reward, total_rewards,
debug=False):
  if debug:
    show_state(env, i_episode, t)
    if done:
        print('')
        if reward == 1:
            total_rewards += 1
            print(">> Success!! got the goal")
        else:
            print(">> Failed!! fell into a hole")
        time.sleep(3), clear_output(wait=True)
    else:
      time.sleep(.5)
    it = max(i_episode, 1)
    print(f"Avg reward so far = {(total_rewards / it):.2f}")
  else:
    if done:
```

```
      if reward == 1:
        total_rewards += 1
      it = max(i_episode, 1)
      if (it+1)%10 == 0:
        print(f"Avg reward obtained in iteration {it} =
{(total_rewards / it):.2f}")
  return total_rewards
```

ACT 7: Call agent, and evaluate results

Below is the main loop of our setting: first, we call the `value_iteration` function, and obtain the optimal policy. Then, we run in many episodes (or trials) of the algorithm. In each trial, we start at the starting state "S". We then ask the agent what action to go towards. We make a step in the frozen lake with the chosen action (but may not move there because the lake is slippery!), our next state is given after we call `env.step(action)`.

You can expect to get an average reward in the range (0.8,1) upon correct implementation.

```
# note that you can first try it out with a random agent, by setting
this variable to True.
random_agent = False
episodes = 100

# When `debug` is set to True, you can see visualizations of the
agent's actions
# While running over 100 episodes, set this to False to speed up your
execution
debug = False

if not random_agent:
    # We have called the value_iteration function:
    V_, pi = value_iteration(gamma, theta)
    # note that we don't need the value function V anymore, as we
already extracted the optimal policy!

total_rewards = 0
for i_episode in range(episodes):
      state = env.reset()
      t = 0
      while True:
          t+=1
          # your agent picks an action:
          if random_agent:
            action = env.action_space.sample() # totally random action
!
          else:
            # ACT 7: get the best action according to optimal policy
`pi`, and current state `state`.
            action = 0
            ### YOUR CODE GOES HERE
```

```
            action = np.nonzero(pi[state])
            action = action[0][0]

          # make a step according to policy
          state, reward, terminated, truncated, info =
env.step(action)
          # the above variables are:
                # state (object): agent's observation, current state
(new state we've reached after the step we took)
                # reward (float) : amount of reward returned after
previous action
                # done (bool): whether the episode has ended
                # terminated (bool): whether a `terminal state` is
reached (only True if the state is the Goal or a Hole!)
                # truncated (bool): whether a truncation condition
outside the scope of the MDP is satisfied (eg. timelimit/agent
physically going out of bounds)
                # info (dict): contains auxiliary diagnostic
information (might be helpful for debugging, though not used in this
assignment)

          done = truncated or terminated # we are done in either case

          # visualization
          total_rewards = pretty_print(i_episode, t, done, reward,
total_rewards, debug=debug)
          if done: break

env.close()
```

```
Avg reward obtained in iteration 9 = 0.78
Avg reward obtained in iteration 19 = 0.79
Avg reward obtained in iteration 29 = 0.83
Avg reward obtained in iteration 39 = 0.85
Avg reward obtained in iteration 49 = 0.86
Avg reward obtained in iteration 59 = 0.88
Avg reward obtained in iteration 69 = 0.90
Avg reward obtained in iteration 79 = 0.91
Avg reward obtained in iteration 89 = 0.90
Avg reward obtained in iteration 99 = 0.91
```

Conceptual Questions (ACTs 8-10)

Notice that in your implementation above, an agent that follows the value-iteration policy performs much better than an agent that follows the random policy (you can run both variations by setting the `random_agent` variable to `True/False`).

Please **briefly** answer the following questions (1-2 sentences per question suffices).

ACT8: *Does the value-iteration policy always obtains reward=1? Explain why/why not.*

Answer:

No, because even the value iteration policy produces the optimal policy, but the probability of reaching the ideal state is not always 1, so if probability is not 1, then the reward also might not always be 1.

ACT9: *Will the random policy always obtain reward=0? Explain why/why not.*

Answer:

No, since it is a random policy, so there is a possiblity that it would randomly select the optimal policy, which would result the reward to be greater than 0.

ACT10: *When averaged over many episodes, will the value-iteration policy always obtain a larger reward than the random policy? Explain why/why not (no need to prove it, just state the appropriate claim shown in lecture and breifly explain how it implies this statement).*

Answer:

The reward of following the optimal policy is greater or equal to the reward of following a random policy, therefore after averaging them over many episodes, the value-iteration policy should obatin a larger reward than the random policy.