CS 4/5789 - Programming Assignment 1

January 31st, 2024

This assignment is due on **February 12, 2025** at 11:59pm.

Section 0: Requirements

This assignment uses Python 3.10+. You will need NumPy. For the visualization (described in section 6), also install Matplotlib. If you are on Linux, you will need to install a gui-backend in order for the visualization to work such as PyQt5. Please run `pip install -e .` in the directory of the `setup.py` before starting the project to make sure the libraries are in the correct version.

It is strongly recommended to use a `conda` environment when working with python in this assignment (and in any capacity). For more information on `conda`, please look at `https://docs.anaconda.com/miniconda/`.

Section 1: Deliverables

For this assignment, we will ask you to work in a 4x4 gridworld. This project consists of the following 5 files

- `setup.py`: This file contains the environment requirements that need to be installed prior to starting the project.
- `visualize.py`: This file can be used to help visualize the V-function and policy. There are also functions that can be used for stepping through value and policy iteration. One can see how to use this file by calling `python visualize.py -h`.
- `MDP.py`: This contains the MDP class which is used to represent our gridworld MDP. This contains the states, actions, rewards, transition probabilities, and the discount factor.
- `test.py`: Test case files. Run this with `python test.py`
- `IH.py`: **Student code goes here.** This file will be used for implementing value iteration, exact policy iteration, and approximate policy iteration.

Please implement all the functions with a **TODO** comment in **IH.py**.

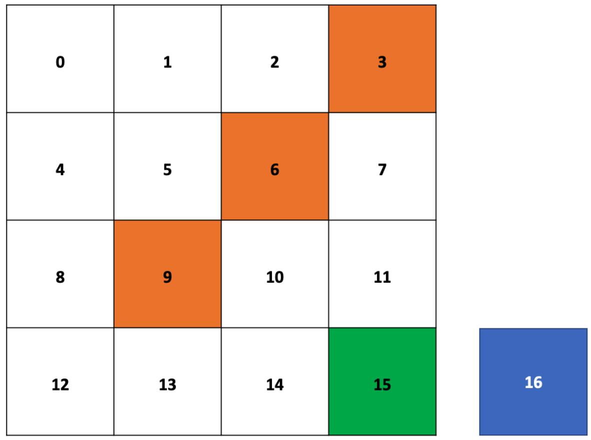
For submission to Gradescope, please do the following:

- Run the visualization for value iteration to iteration 20 with `gamma=={0.9}`. Save the figure showing the Q-values. Additionally, save the figure showing the max policy and its corresponding value function for `gamma=={0.5,0.9,0.999}`.
- Run the visualization for `gamma=={0.9}` for exact and iterative policy iteration for 5 iterations. Save both figures.
- Create a PDF writeup in which you include the figures above and the answers to the questions in the discussion section 7.
- Zip the PDF writeup along with the 5 python files listed above into `submit.zip` and submit on Gradescope.

Section 2: Understanding the MDP

2.1 States and rewards

There are 17 (0-16) states where each box corresponds to a state as shown below



State 15 (the green box) shown above is the "goal" state and has a reward of 200. All the white boxes have reward -1 while the orange boxes are "bad" states that have -80 reward. While our gridworld is 4x4, we need an extra state (state 16) to be the terminal state. This state has reward 0 and is absorbing which means once we enter the terminal state, we cannot leave as any action takes us back to the terminal state. Additionally, once we hit the goal state, we automatically transition to the terminal state. This terminal state allows us to model tasks which terminate at a finite time using the infinite horizon MDP formulation.

Note that the rewards here are outside the range [0,1]. Although many theoretical results assume the reward lies within [0,1], in practice the reward can be any scalar value.

Since we are using a simple state representation, we only need to store the number of states in the MDP class. The rewards, R, is a $|A| \times |S|$ numpy array (note the first index is the action unlike in class). That is, r(s,a) = R[a,s]. The rewards in this MDP are deterministic.

2.2 Actions

There are 4 actions that can be taken in each state. These actions are up (0), down (1), left (2), right (3).

2.3 Transition probabilities

The transition probabilities are easy to define for a gridworld MDP which makes it a good setting for implementing value iteration, policy evaluation, and policy iteration. The transition matrix P, in `MDP.py`, contains the transition probabilities. Note that the transitions are stochastic (i.e. moving right from state 0 isn't guaranteed to move you to state 1, you may "slip" and move to state 4 or stay in the same spot). In particular, when transitioning from a white or orange box, the agent ends up in the correct spot with probability 0.7, and slips laterally with probability 0.15 to either of the adjacent spots.

P is an $|A| \times |S| \times |S|$ NumPy array. The transition probability $P(s'|s,a)$ can be found by indexing $P[a,s,s']$

2.4 Policies

We will work with deterministic policies and will represent policies using a NumPy array of size $|S|$. The s^{th} entry in such a vector represents the (deterministic) action taken by the policy at state s.

Section 3: Policy Evaluation

We will now switch gears and implement the algorithms seen in class. In this section, we ask you to implement both exact policy evaluation and iterative policy evaluation.

3.1 TODOs

For value iteration, there are 2 functions (TODOs) to complete.

- `exactPolicyEvaluation`
- `approxPolicyEvaluation`

3.2 Exact Policy Evaluation

The Bellman equation for deterministic policies, π, from class is as follows

$$\begin{aligned} V^{\pi}(s) &= r(s, \pi(s)) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,\pi(s))} V^{\pi}(s') \\ &= r(s, \pi(s)) + \gamma \sum_{s'} \mathbb{P}(s'|s, \pi(s)) V^{\pi}(s') \end{aligned}$$

Using vector notation, this can be written as the linear equation

$$V = R^{\pi} + \gamma P^{\pi} V$$

Hint: Use `extractRpi` and `extractPpi` to easily extract $r(s, \pi(s))$ and $P(s'|s, \pi(s))$ in vector form.

By solving this, we can compute V^{π}. Do **not** use matrix inversion for this. Instead use `np.linalg.solve`.

Section 4: Value Iteration with the Q-function

4.1 TODOs

For value iteration, there are 6 functions (TODOs) to complete.

- `computeVfromQ`
- `computeQfromV`
- `extractMaxPifromQ`
- `extractMaxPifromV`
- `valueIterationStep`
- `valueIteration`

4.2 Switching between Q and V functions

Given V^π and Q^π, please implement `computeVfromQ` and `computeQfromV` using the equations

$$\begin{aligned} Q^\pi(s,a) &= r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} V^\pi(s'), \\ V^\pi(s) &= Q^\pi(s, \pi(s)). \end{aligned}$$

4.3 Computing max policy

From class, we know that if we have the optimal Q-function, Q^*, we can find the optimal policy

$$\pi^\star(s) = \arg\max_a Q^\star(s)$$

We can also use this operation at the end of value iteration on Q^t (our final Q values) and also during the policy improvement stage of policy iteration. Please implement `extractMaxPifromQ` and `extractMaxPifromV`.

4.4 Value Iteration

First, implement `valueIterationStep`. This implements the following equation

$$\forall s, a : Q^{t+1}(s,a) = r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \max_{a'} Q^t(s', a')$$

Q^t is passed in to this function while the rewards and transition probabilites can be accessed from the attributes. While not strictly necessary, try implementing this function without for-loops.

Next, implement `valueIteration` following the specification. This should consist of running `valueIterationStep` in a loop until the threshold is met. Then, extract the policy from the final Q-function and compute the corresponding V-function. Return the policy, V-function, iterations required until convergence, and the final epsilon.

4.5 Iterative Policy Evaluation

Implement approximate policy evaluation following the algorithm given in class. Each step of iterative policy evaluation involves computing V^{t+1} given the current estimate V^t

$$V^{t+1} = R^\pi + \gamma P^\pi V^t$$

Run this algorithm until convergence (as defined by the tolerance).

Section 5: Policy Iteration

Finally, we will move onto policy iteration. This is another iterative algorithm in which we cycle between policy evaluation and policy improvement to continuously improve our policy. Note the difference between this algorithm and value iteration. In value iteration, we did not compute a policy until we computed the final estimate of the optimal Q-function.

5.1 TODOs

For policy iteration, there are 2 functions (TODOs) to complete.

- `policyIterationStep`
- `policyIteration`

First, implement the function `policyIterationStep`. Given the current policy π^t, compute the value-function for π^t using policy evaluation. This can be done using either exact or iterative policy evaluation. With V^{π^t}, compute the new policy.

Once `policyIterationStep` is implemented, implement `policyIteration` in the same style as `valueIteration`. That is, continuously call `policyIterationStep` until convergence. Convergence in this case occurs when the policy improvement step does not change our current policy.

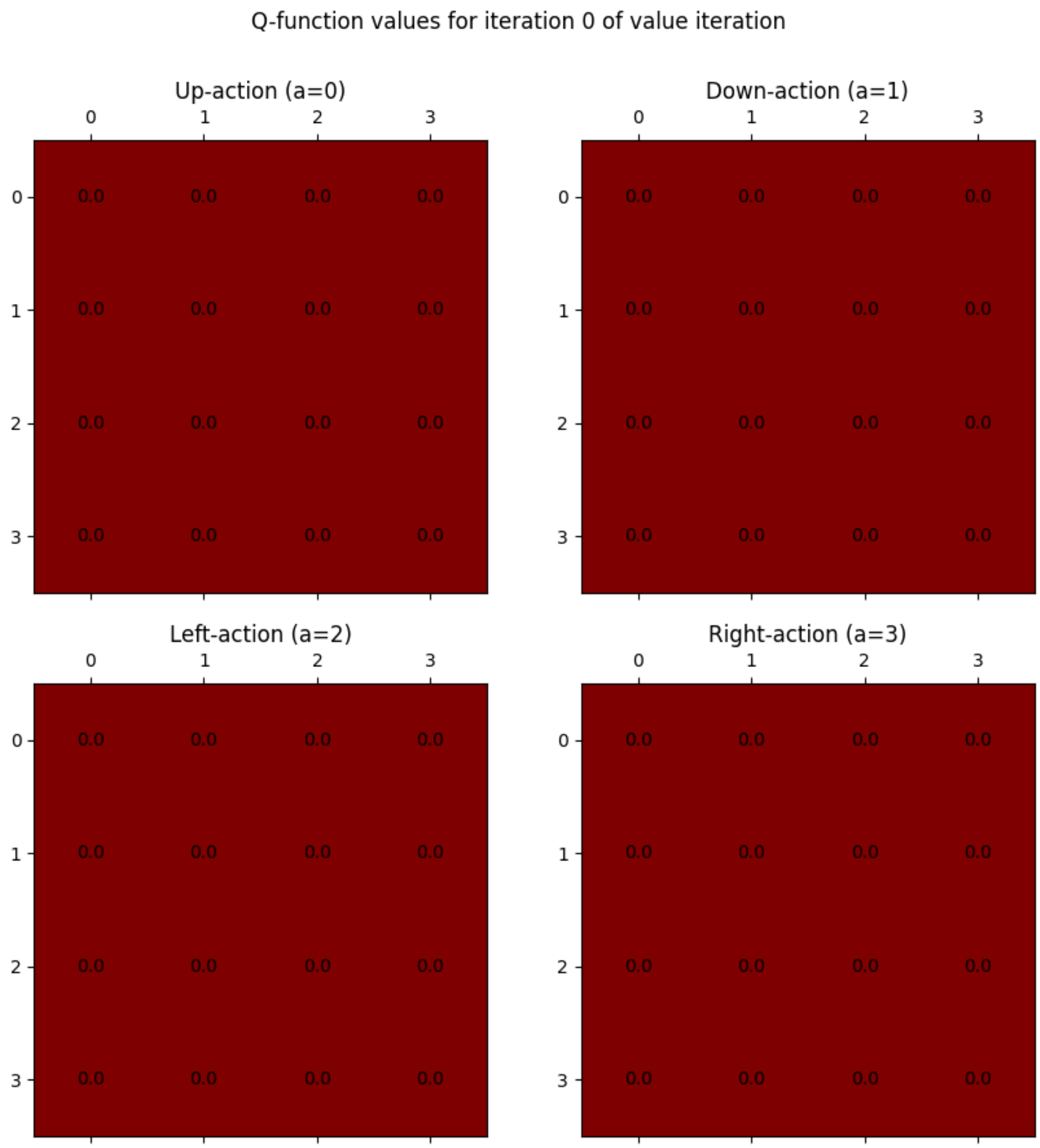
Section 6: Testing & Visualization

We have written some basic tests for you to test your implementation. These are separated into tests for DP, VI, PE, PI. Please note that these tests are basic and do not guarantee that your implementation is correct. We have also given you code for visualizing value iteration and policy iteration.

6.1 Value Iteration

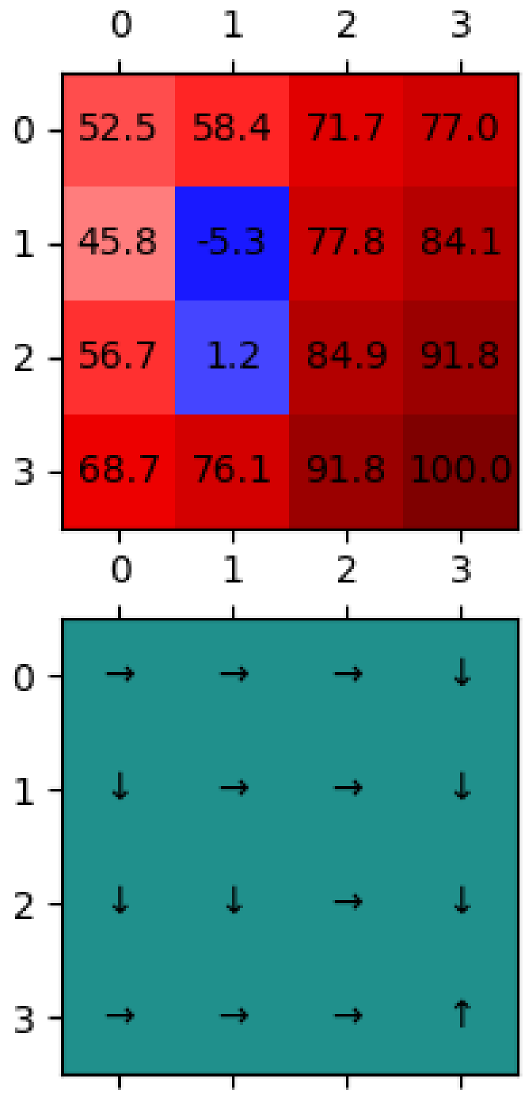
After implementing value iteration, run `python visualize.py --alg VI --gamma 0.95`. This will allow you to visualize value iteration from an initial Q^0 of all zero's. To change the initial Q^0, change line 291 in `visualize.py`.

Running this command should open the following window.



This shows the Q-values for each of the 4 actions. To step through an iteration of value iteration, press the **right** arrow key. The grids should update as you step through value iteration.

Normally, value iteration runs until convergence and then the final policy is extracted. However, we have added extra visualization to allow you to see how the policy changes. For any iteration, t, press the **enter** key to compute $\pi(s) = \arg\max_a Q^t(s, a)$ and V^π. A new window should show up showing the V-function and policy as shown below



To continue stepping through value iteration, exit this window and continue pressing the **right** arrow key on the main figure window.

6.2 Policy Iteration

To visualize policy iteration, run `python visualize.py --alg PI_exact --gamma 0.95` or run `python visualize.py --alg PI_approx --gamma 0.95`. Again, use the **right** arrow key to step through policy iteration. To change π^0, change line 171 or 174 of `visualize.py`.

Section 7: Discussion

Please answer the following questions in a few sentences and include your answers in the writeup.

- Let's consider the infinite horizon problem. Compare the value function and optimal policy for different values of γ. How does it change? Why?