Lab 2
Dynamic Programming:
Policy iteration and Value Iteration

September, 2025

1 Goal of the Lab

In this lab you will implement dynamic programming methods to learn to evaluate policies in an MDP setting, and also to find the optimal policy for given an MDP, and you will use them to solve the `RussellGrid` environment we presented in the last lab.

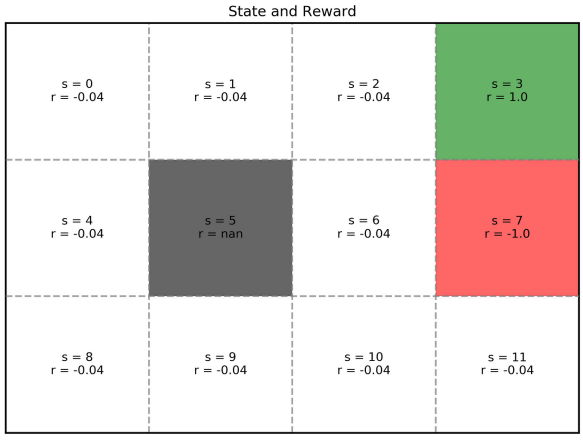

Figure 1: Russell's Grid: Green and Red cells are terminal states (trial ends when arriving to these states) and black state is impossible to achieve (consider it as a column in the room). The dynamics of the world consists of 4 possible actions (Up, Right, Down, Left) that move you in the intended direction with probability 0.8 but fail with probability 0.2 leading to a random adjacent cell. When running against a wall or the column in the room, the agent remains in the original state.

Input π, 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 2: Policy Evaluation.

2 Programming Tasks:

In the `lab2.zip` file you will find a new version of the Grid.py file that implements the `RussellGrid` as an MDP. The environment has been extended with the attribute `P` that consists in a dictionary where you can pass a state and an action, and returns a list with pairs of possible next_states and the probability of going there. For instance:

```
env.P[state][action]
```

where state is (0,0) and action is 1 (that means RIGHT) will return the list

```
[(0.8, (0, 1)), (0.1, (0, 0)), (0.1, (1, 0))]
```

that means you have probability 0.8 to go to state (0,1), probability 0.1 to remain in state (0, 0)), and probability 0.1 to go to state (1, 0).

Remember that you also have `env.reward_matrix[state]` that returns the reward for that state.

Your task will consist in complete the code of the following functions[1]:

1. **Policy evaluation (one step):** Inside `policy_evaluation_one_step`, fill in the code to perform exactly one step of policy evaluation (the inner loop of the algorithm inside figure 2).

2. **Policy evaluation:** Using `policy_evaluation_one_step`, implement the method `policy_evaluation` (which corresponds to the whole algorithm of figure 2) which iteratively performs one step of policy evaluation until the changes in the value function estimate are smaller than some predefined threshold.

[1]The file to be completed can be either `DynamicProgramming.py` (a regular python file to run in local), or `DynamicProgramming.ipynb` (a notebook you can run in Google Colab) that you will find inside `lab2.zip`.

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
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      $a \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      If $a \neq \pi(s)$, then *policy-stable* $\leftarrow false$
   If *policy-stable*, then stop and return V and π; else go to 2

Figure 3: Policy 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, π, such that
   $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Figure 4: Value Iteration.

3. **Policy improvement:** Continue with the `policy_improvement` function (see step 3 of figure 3), which returns the optimal greedy policy w.r.t a given value function. To ease the writing of the function, I recommend you to implement and use a function named `greedy_action` that returns the best action to do on a given state and for a given value function (it would implement line 6 of step 3 of figure 3).

4. Finally, utilizing your results from 1. - 3. implement:

   (a) the policy iteration (PI) algorithm (figure 3) inside the function `policy_iteration`. You can use the function `print_policy(policy)` at each iteration to see how the policy evolves, and finally you get the optimal policy. The function is defined in the file `utils.py`.

   (b) the value iteration (VI) algorithm (figure 4) inside the function `value_iteration`. At the end of the function, you can call the function `print_policy(policy)` to check that the policy found is the optimal one. If you call `print_policy(policy,V)` with the value function, it will paint with colors in the map the values V(s) of each state.