

## Problem 1: Markovian Drone

In this problem, we will apply techniques for solving a Markov Decision Process (MDP) to guide a flying drone to its destination through a storm. The world is represented as an  $n \times n$  grid, i.e., the state space is

$$\mathcal{X} := \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1, x_2 \in \{0, 1, \dots, n-1\}\}.$$

In these coordinates,  $(0, 0)$  represents the bottom left corner of the map and  $(n-1, n-1)$  represents the top right corner of the map. From any location  $x = (x_1, x_2) \in \mathcal{X}$ , the drone has four possible directions it can move in, i.e.,

$$\mathcal{U} := \{\text{right}, \text{up}, \text{left}, \text{down}\}.$$


The corresponding state changes for each action are:

- **right:**  $(x_1, x_2) \mapsto (x_1 + 1, x_2)$
- **up:**  $(x_1, x_2) \mapsto (x_1, x_2 + 1)$
- **left:**  $(x_1, x_2) \mapsto (x_1 - 1, x_2)$
- **down:**  $(x_1, x_2) \mapsto (x_1, x_2 - 1)$

Additionally, there is a storm centered at  $x_{\text{eye}} \in \mathcal{X}$ . The storm's influence is strongest at its center and decays farther from the center according to the equation  $\omega(x) = \exp\left(-\frac{\|x - x_{\text{eye}}\|_2}{2\sigma^2}\right)$ . Given its current state  $x$  and action  $u$ , the drone's next state is determined as follows:


- With probability  $\omega(x)$ , the storm will cause the drone to move in a uniformly random direction.
- With probability  $1 - \omega(x)$ , the drone will move in the direction specified by the action.
- If the resulting movement would cause the drone to leave  $\mathcal{X}$ , then it will not move at all. For example, if the drone is on the right boundary of the map, then moving right will do nothing.

The quadrotor's objective is to reach  $x_{\text{goal}} \in \mathcal{X}$ , so the reward function is the indicator function  $R(x) = I_{x_{\text{goal}}}(x)$ . In other words, the drone will receive a reward of 1 if it reaches the  $x_{\text{goal}} \in \mathcal{X}$ , and a reward of 0 otherwise. The reward of a trajectory in this infinite horizon problem is a discounted sum of the rewards earned in each timestep, with discount factor  $\gamma \in (0, 1)$ . Take a look at `Problem_1/value_iteration.py`.


- (i)  Given  $n = 20$ ,  $\sigma = 1$ ,  $\gamma = 0.95$ ,  $x_{\text{eye}} = (15, 7)$ , and  $x_{\text{goal}} = (19, 9)$ , write code that uses value iteration to find the optimal value function for the drone to navigate the storm. Recall that value iteration repeats the Bellman update

$$V(x) \leftarrow \max_{u \in \mathcal{U}} \begin{cases} R(x, u) + \gamma \sum_{x' \in \mathcal{X}} p(x'; x, u) V(x') & \text{if } x \text{ is not a terminal state} \\ R(x, u) & \text{otherwise} \end{cases}$$

until convergence, where  $p(x'; x, u)$  is the probability distribution of the next state being  $x'$  after taking action  $u$  in state  $x$ , and  $R$  is the reward function.

- (ii)  Plot a heatmap of the optimal value function obtained by value iteration over the grid  $\mathcal{X}$ , with  $x = (0, 0)$  in the bottom left corner,  $x = (n-1, n-1)$  in the top right corner, the  $x_1$ -axis along the bottom edge, and the  $x_2$ -axis along the left edge.


*Hint:* We provide a function that plots the heatmap: `visualize_value_function()`.

- (iii)  Recall that a policy  $\pi$  is a mapping  $\pi : \mathcal{X} \rightarrow \mathcal{U}$  where  $\pi(x)$  specifies the action to be taken should the drone find itself in state  $x$ . An optimal value function  $V^*$  induces an optimal policy  $\pi^*$  such that

$$\pi^*(x) \in \operatorname{argmax}_{u \in \mathcal{U}} \left\{ R(x, u) + \gamma \sum_{x' \in \mathcal{X}} p(x'; x, u) V^*(x') \right\}$$

Note that the optimal policy is only defined for non-terminal states.

Use the value function you computed in part (a) to compute an optimal policy. Then, use this policy to simulate the MDP starting from  $x = (0, 0)$  over  $N = 100$  time steps.


- (iv)  Plot the policy as a heatmap. Plot the simulated drone trajectory overlaid on the policy heatmap, and briefly describe in words what the policy is doing.  
*Hint: Feel free to modify `visualize_value_function()` or write it from scratch yourself.*

Another popular approach to reinforcement learning is to use a Q-network which encodes expected future discounted reward for both a given state and action pair. The optimal policy for a value function is given by:

$$\pi^*(x) \in \operatorname{argmax}_{u \in \mathcal{U}} Q(x, u)$$

A very popular approach is to approximate the Q-function as a feed-forward neural network. We will now prepare training data, train the Q-function approximation (or the Q-network now) and compare the approximately optimal policy to the optimal policy found by value iteration.


Take a look at `Problem_1/q_learning.py`.

- (v)  Given the same environment, sample  $10^5$  state transition triples

$$(x_i, u_i, x'_i) \quad \forall i \in [1, 10^5]$$

Make sure to generate transition samples with appropriate probability of getting blown off course by the storm.



**Important: We are representing the state as a two dimensional vector with the two grid coordinates and the action as a single element vector.** You are free to use your own state and action representation, but another choice might require a significant amount of tuning of the resulting Q-network.

- (vi)  In order to develop the Q-network loss function, we will start by writing down the Bellman equation of the optimal Q-function—our Q-network should aim to approximate that. Write down the expectation form of the optimal Q-function in terms of an equality

$$Q^*(x, u) = \dots \tag{1}$$

*Hint: Your form should not contain the transition probabilities, since we do not know those.*


*Hint: Remember to account for the reward value and whether the state is terminal.*

- (vii)  Create a feed forward neural network for representing the Q-network. Use 3 dense layers with a width of 64 (two hidden 64 neuron embeddings).
- (viii)  Train the neural network filling in the provided `Q_learning` Python function. Use the Adam optimizer. Experiment with the following step sizes  $\{10^{-3}, 10^{-2}, 10^{-1}\}$  and pick the one that works best.


We will now develop the Q-network loss as an L2 penalized equality (1) residual.

$$\ell = \frac{1}{n} \sum_{i=1}^n \|\text{LHS}_i - \text{RHS}_i\|_2^2 \quad \forall i \in S_{\text{examples}}$$

*Hint: The expectation operator can be silently dropped since we are (1) using a quadratic residual penalty and (2) summing over all of the samples—which is equivalent to taking empirical expectation over data.*

- (ix)  Describe a dynamical system or a dataset situation in which using Q-learning could be easier than using value iteration.

*Hint: You don't need to limit the example to the field of robotics.*

- (x)  Include a binary heatmap plot that shows, for every state, if the approximate Q-network policy agrees or disagrees with the value iteration optimal policy.

*Hint: Feel free to modify `visualize_value_function()` or write it from scratch yourself.*