CSCE 580: Artificial Intelligence
Coding Project 4
Due: 04/20/2021 at 11:59pm

Installation

We will be using the same `conda` environment as in Homework 0.

The entire GitHub repository should be downloaded again as changes were made to other files. You can download it with the green "Code" button and click "Download ZIP".

1 Policy Iteration

Key building blocks:

- `env.state_action_dynamics(state, action)`: returns, in this order, the expected reward $r(s, a)$, all possible next states given the current state and action, their probabilities. Keep in mind, this only returns states that have a non-zero state-transition probability.
- `env.get_actions()` function that returns a list of all possible actions
- `policy`: you can obtain $\pi(a|s)$ with `policy[state][action]`

Switches:

- `--discount`, to change the discount (default=1.0)
- `--rand_right`, to change the probability that the wind blows you to the right (default=0.0)
- `--wait`, the number of seconds to wait after every iteration of policy *iteration* so that you can visualize your algorithm (default=0.0)

1.1 Policy Evaluation Implementation (30 pts)

Implement `policy_evaluation_step` (shown in Algorithm 1) in `proj_code/proj4.py`.

Running the code:
`python run_policy_iteration.py --map maps/map1.txt --wait 1.0`

Policy iteration can be used to compute the optimal value function. Policy iteration starts with a given value and policy function and iterates between policy evaluation and policy improvement until the policy function stops changing. For more information, see the lecture slides or Chapter 4 of Sutton and Barto. In this exercise, we will be finding the optimal value function using policy iteration.

We will start with a uniform random policy and a value function that is zero at all states.
Vary the environment dynamics by making the environment stochastic `--rand_right` to 0.1 and 0.5.
Compare your results to the solution videos provided in Blackboard under Course Content - ProjectExamples - Project4.

Algorithm 1 Policy Evaluation

procedure POLICY EVALUATION($\mathcal{S}, V, \pi, \gamma$)
  $\Delta \leftarrow \inf$
  while $\Delta > 0$ **do**
    $\Delta \leftarrow 0$
    for $s \in \mathcal{S}$ **do**
      $v \leftarrow V(s)$
      $V(s) \leftarrow \sum_a \pi(a|s)(r(s,a) + \gamma \sum_{s'} p(s'|s,a)V(s'))$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
  end while
  return V ▷ v_π
end procedure

1.2 Policy Improvement (20 pts)

Implement `policy_improvement` (shown in Algorithm 2) in `proj_code/proj4.py`. Remember that the policy that you return will map states to the probability of each action. Since the policy is deterministic, only one action should have a non-zero probability with a value of 1.

Algorithm 2 Policy Improvement

procedure POLICY IMPROVEMENT($\mathcal{S}, V, \gamma$)
  for $s \in \mathcal{S}$ **do**
    $\pi'(s) = \text{argmax}_a(r(s,a) + \gamma \sum_{s'} p(s'|s,a)V(s'))$
  end for
  return π'
end procedure

2 Q-learning (50 pts)

Implement `q_learning_step` (shown in Algorithm 4) in `proc_code/proj4.py`.

Key building blocks:

- `env.sample_transition(state, action)`: returns, in this order, the next state and reward
- `env.get_actions()` function that returns a list of all possible actions
- `action_vals`: you can obtain $Q(s,a)$ with `action_vals[state][action]`

Switches:

- `--discount`, to change the discount (default=1.0)
- `--learning_rate`, to change the discount (default=0.5)
- `--epsilon`, to change the discount (default=0.1)
- `--rand_right`, to change the probability that the wind blows you to the right (default=0.0)

- `--wait`, Your learned greedy policy is visualized every 100 episodes for 40 steps. This is the number of seconds to wait after each step (default=0.1)
- `--wait_step`, the number of seconds to wait after every step of Q-learning so that you can visualize your algorithm (default=0.0)

Running the code:
`python run_q_learning.py --map maps/map1.txt --wait_greedy 0.1`

Q-learning is a model-free, off-policy, temporal difference algorithm. Q-learning follows an ϵ-greedy behavior policy and evaluates a greedy target policy. An ϵ-greedy policy takes a random action with probability ϵ, otherwise it takes the greedy action, $\text{argmax}_a\, Q(S, a)$.

In this setting, we will be running Q-learning with a learning rate $\alpha = 0.5$ and $\epsilon = 0.1$. Each episode ends when the agent reaches the goal. We will run Q-learning for 1000 episodes. For more information, see the lecture slides or Chapters 5 and 6 of Sutton and Barto.

Algorithm 3 Q-learning

procedure Q-LEARNING($Q, \gamma, \epsilon, \alpha$)
  for $i \in 1...N$ **do**
    Initialize S
    while S is not terminal **do**
      $S, Q =$ Q_Learning_Step($S, Q, \epsilon, \alpha, \gamma$)
    end while
  end for
  return Q ▷ Approximation of q_*
end procedure

Algorithm 4 Q-learning Step

procedure Q_LEARNING_STEP($S, Q, \epsilon, \alpha, \gamma$)
  Sample an action A from ϵ-greedy policy derived from Q
  $S', R =$ `env.sample_transition(S,A)`
  $Q(S, A) = Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$
  return S', Q
end procedure

Vary the environment dynamics by making the environment stochastic with `--rand_right` to 0.1 and 0.5.

Compare your results to the solution videos provided in Blackboard under Course Content - ProjectExamples - Project4. Keep in mind that this algorithm is stochastic, so results will not be exactly the same.

Important: since Q-learning is a model-free algorithm, you cannot use `env.state_action_dynamics` in your implementation.

What to Turn In
Turn in your implementation of to `proj_code/proj4.py` to Coding Projects/Project 4 on Blackboard.