CSCE 790-002: Deep Reinforcement Learning and Search
Homework 2
Due: 09/30/2020

Installation

We will be using the same `conda` environment as in Homework 1.

Keep in mind that this assignment relies on a correct implementation of `assignments_code/assignment1.py`, which is provided on Blackboard.

1 Policy Iteration

1.1 Policy Iteration Implementation (30 pts)

Implement `policy_evaluation_step` (shown in Algorithm 2) in `assignments_code/assignment2.py`.

Key building blocks:

- `env.state_action_dynamics(state, action)`: returns, in this order, the expected return $r(s, a)$, 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)
- `--wait_eval`, the number of seconds to wait after every iteration of policy *evaluation* so that you can visualize your algorithm (default=0.0)

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

Policy iteration can be used to find, or approximate, 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. In this exercise, we will be finding the optimal value function, exactly, using policy iteration.

We will start with a uniform random policy and a value function that is zero at all states. The policy improvement step has already been implemented using the `get_action` function that you implemented in Homework 1. Feel free to use your current implementation or the solution provided to you.
Therefore, you only have to implement `policy_evaluation_step`.

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 - Homework - Homework 2.

Algorithm 1 Policy Evaluation

procedure POLICY EVALUATION($\mathcal{S}, V, \pi, \gamma, \theta$)
    $\Delta \leftarrow \inf$
    while $\Delta > \theta$ **do**
        $\Delta, V = \text{Policy\_Evaluation\_Step}(\mathcal{S}, V, \pi, \gamma)$
    end while
    return V ▷ Approximation of v_π
end procedure

Algorithm 2 Policy Evaluation Step

procedure POLICY_EVALUATION_STEP($\mathcal{S}, V, \pi, \gamma$)
    $\Delta \leftarrow 0$
    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
    return Δ, V
end procedure

1.2 Policy Iteration Concept (20 pts)

For policy evaluation, we started with a uniform random policy, a discount of $\gamma = 1$, and stopped policy evaluation only when $\Delta = 0$. If we started, instead, with a policy that gave the same action in every state (i.e. in every state, go left) we would never converge to $\Delta = 0$. Why is this? If we reduce γ, we would converge to $\Delta = 0$. Why is this?

2 Q-learning

2.1 Q-learning Implementation (30 pts)

Implement `q_learning_step` (shown in Algorithm 4) in `assignments_code/assignment2.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)

- `--rand_right`, to change the probability that the wind blows you to the right (default=0.0)
- `--wait_greedy`, 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 but evaluates a greedy policy. 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 about ϵ-greedy policies, please refer to 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
    return Q ▷ Approximation of q_*
  end for
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 - Homework - Homework 2. Keep in mind that this algorithm is stochastic, so results will not be exactly the same.

2.2 Q-learning Concept (20 pts)

In the case where we do Q-learning with `--rand_right` set to 0.5, the actions in the states in the top left remain bright green, indicating that these states are, relatively, the best states. However, we know from finding the optimal value function via policy iteration that this is not the case. Why does this occur and why do we not run in to this problem when `--rand_right` set to 0.5 when doing value iteration or policy iteration?

What to Turn In

Turn in your implementation of `assignments_code/assignment2.py` and a PDF of your answer to question parts 1.2 and 2.2.