CSCE 790-001: Deep Reinforcement Learning and Search
Coding Homework 1
Due: 09/15/2021 at 11:59pm

Your code must run in order to receive credit.
For grading purposes, your code must return exactly what is specified in the documentation.

To check your implementations vary the environment dynamics by making the environment stochastic versions of the environment such as `aifarm_0.1` and `aifarm_0.5`.

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 Dynamic Programming

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
- `state_values`: you can obtain $V(s)$ with `state_values[state]`
- `policy`: you can obtain $\pi(a|s)$ with `policy[state][action]`
- `viz`: you can use `update_dp(viz, state_values, policy)` to visualize your algorithm. You can modify method signatures to pass this to other functions, such as policy evaluation, as long as you return the method signatures back to their original state when you turn in your work.

Switches:

- `--env`, environment name. AI farm is `aifarm_<prob>` where `<prob>` is the probability that the wind blows you to the right. For example `aifarm_0` is deterministic and `aifarm_0.1` is stochastic.
- `--discount`, to change the discount (default=1.0)

1.1 Policy Iteration (50 pts)

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 and Chapter 4 of Sutton and Barto. In this exercise, we will be finding the optimal value function using policy iteration. Terminate policy iteration when the policy stops changing. In your implementation, return the state values and the policy (in that order) found by policy iteration.

Implement the following functions in `code_hw/code_hw1.py`: `policy_evaluation`, `policy_improvement`, and `policy_iteration`.

Running the code:
```
python run_code_hw_1.py --env aifarm_0 --algorithm policy_iteration
```

The policy starts as a uniform random policy and a value function that is zero at all states.

1.2 Value Iteration (25 pts)

Value iteration can be seen as combining the policy evaluation and policy improvement step into a single step. Value iteration starts with a given value function and stops when the change between the previous value function and updated function falls below a given threshold. For more information, see the lecture slides and Chapter 4 of Sutton and Barto. In this exercise, we will be finding the optimal policy using value iteration. In your implementation, return the state values and the policy (in that order) found by value iteration.

Implement the following function in `code_hw/code_hw1.py`: `value_iteration`.

Running the code:
```
python run_code_hw_1.py --env aifarm_0 --algorithm value_iteration
```

2 Q-learning Learning (25 pts)

Q-learning is an off-policy model-free reinforcement learning algorithm. For more information, see the lecture slides and Chapters 5 and 6 of Sutton and Barto.

Important: In this exercise, you will be implementing Q-learning and we will assume that we do not have access to the dynamics of the MDP. Therefore, in your implementation of Q-learning, you cannot use `env.state_action_dynamics(state, action)`.

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
- `env.sample_start_state()` function that returns a start state
- `action_vals`: you can obtain $Q(s, a)$ with `action_vals[state][action]`

Switches:

- `--env`, environment name. AI farm is `aifarm_<prob>` where `<prob>` is the probability that the wind blows you to the right. For example `aifarm_0` is deterministic and `aifarm_0.1` is stochastic.
- `--discount`, to change the discount (default=1.0)
- `--epsilon`, to change the ϵ for the ϵ-greedy policy (default=0.1)

- `--learning_rate`, to change the learning rate (default=0.5)
- `viz`: you can use `update_model_free(viz, state, action_values)` to visualize your algorithm.

Implement the following function in `code_hw/code_hw1.py`: `q_learning`.

Running the code:

```
python run_code_hw_1.py --env aifarm_0 --algorithm q_learning
```

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 or when the number of steps taken reaches 50. We will run Q-learning for 1000 episodes.

What to Turn In

Turn in your implementation of to `code_hw/code_hw1.py` to Homework/Coding Homework 1 on Blackboard.