Homework 4

Apply Dynamic Programming and Reinforcement Learning to the well-known game – Pacman.

****

The GUI of this game is modified from the example code

of a python third-party module “freegames.”

It is required that you use Python 3 for this homework and with freegames installed.

(pip install freegames or pip3 install freegames)

It is highly recommended that you use the same Python version as me (3.7).

The code might be compatible with some versions prior to 3.7, but I’m not sure how far it can go back to.

****

The following command-line instructions are under the assumption that you install Python in a way that you use “python xxx.py” to run a python file.

Depending on your installation configuration, you might actually need to use “python3 xxx.py” or something else.

- In this homework, you need to implement some methods in IntelligentAgents.py with the following algorithms so that your pacman can beat the ghost every time.

q1. Dynamic Programming – Value Iteration

q2. Dynamic Programming – Policy Iteration

q3. Reinforcement Learning – Approximate Q-Learning

By running “python autograder.py -q qx” to see the score of question x.

Or simply “python autograder.py” to see all the scores.

- Next, the base settings of the game will be introduced, and then details of each question will be described.

- So, how do we model this game into an MDP?

An MDP is characterized by a set of possible states ({s}), sets of actions valid for the states ($\{a\}_s$), transition probabilities (T(s, a, s')), and rewards of corresponding transitions (R(s, a, s')).

States: the positions of pacman and ghost, whether the food is eaten for each food.

Actions: {"up", "down", "left", "right"} without resulting in an invalid position.

> In the code of an Agent, use "self.valid_moves(pos, self.game.valid)" (where pos is the current position of the agent) to get such valid actions.

Transition Probabilities:

> This is the most important part, because in normal case, this game is deterministic, which means it cannot be an MDP. So for this homework, we make the ghost's action to be "noisy", which means it cannot just take the action it wants to take, but with some probability it could go other directions. As a result, we assume that, the game proceeds by "one pacman move, one ghost move", and after the pacman moves, the resulted state is treated as a Q-state. When it's the ghost's turn to take an action, due to the noise, there will be several possible next states (s') with well-defined probability. About the probability values and possible resulted states, see the comments in the code for more details. (IntelligentAgents.py/MDPAgent.probability)

Rewards:

> We simply define the reward as the difference of scores between current state and next state (stored in game state).

- There are three Game-related classes, Game, ValueIterGame, and LearningGame defined in correspondingly-named python files.

I'll introduce what are stored in the game state data structure of each class and helpful methods that you need for implementing the algorithms.

There's one more thing, we use a class named "vector" from freegames to represent positions and moves. The API of it is here: http://www.grantjenks.com/docs/freegames/api.html#vectors. It can do anything what you expect a vector should be able to do, like vector + vector, vector * number, and more.

Class *Game*:

Handling the game rules, flows, and state changes.

The methods you can use:

1. *Game*.valid(pos<vector>) -> valid?<bool>

   @pos: the position to test.

   @Return valid?: whether a position is valid (not in a wall) in the tiles of this game.

2. *Game*.end_game() -> end_type<int>

   # you will only need to use the overridden version of it by the derived classes

3. *Game*.nextstate(agent, param1, param2) -> nextstate<dict>

   # you will only need to use the overridden version of it by the derived classes

Run "python Game.py" to play Pacman with keyboard **just for fun**.

(It is difficult to win for me, see if you can beat the ghosts.)

Class *ValueIterGame*:

Specialized *Game* class for the two Dynamic Programming algorithms. (q1 & q2)

Dynamic Programming requires all the possible states and their values to be computed.

But the number of possible states (we call it state space size) grows very quickly as the grid size or the total food number grows.

Thus, we use a relatively much smaller grid world with fewer ghosts and less food than *Game* to reduce the state space size.

(The state space size here is:

24 (pacman position) x 24 (ghost position) x 2^4 (whether a food is eaten) = 9216)

In the game state (we use a dictionary for implementation), the following information is included:

"pacman_pos": position of pacman <vector>

"ghost1_pos": position of ghost <vector>

"food_num": number of foods not eaten yet <int>

"foods": positions of non-eaten foods <list<vector>>

"score": game score in such state

(get +1 for each food eaten, +100 for winning, and -200 for losing.)

The methods you can use:

1. *ValueIterGame*.end_game(state<dict> = None) -> end_type<int>

   @state: state to test if ended, if not given, use *ValueIterGame*.state

   @Return end_type: condition of the state, 0 for not ended, 1 for losing, and 2 for winning.

2. *ValueIterGame*.nextstate(agent<string>, param1, param2) -> nextstate<dict>

   This method is meant for you to "peek" at the Q state and next state from some state while planning.

   In Dynamic Programming, we call it planning rather than learning, because it knows everything (possible states, rewards, transition probabilities, and so on) in advance.

   Before it really starts playing the game, it can already construct the policy and just takes action according to the current state and the policy while playing.

   Therefore, it wouldn't really change the game state nor move the agents in this method, but return a copied dictionary representing the resulted state.

   For this class, there are two possible agents: "sim_pac" and "sim_ghost".

   "sim_pac": @param1: action<vector>, @param2: "from" state<dict>

   @Return the resulted state by moving the pacman in state["pacman_pos"] with the given action.

   "sim_ghost": @param1: action<vector>, @param2: "from" Q-state<dict>

   @Return the resulted state by moving the ghost in Q-state["ghost1_pos"] with the given action.

Run “python ValueIterGame.py -p key” to play with keyboard.

“python ValueIterGame.py -p value” to plan and play with value iteration

“python ValueIterGame.py -p policy” to plan and play with policy iteration

If you run the above with one more command-line argument “-e”, you will see your agent fight a non-noisy ghost and I promise, it won’t win.

“DP.mp4” shows what it should look like if your code is correct.

Class *LearningGame*:

Specialized *Game* class for the Reinforcement Learning algorithms. (q3)

Unlike Dynamic Programming, Reinforcement Learning only stores and computes values of states that the agent has ever experienced. As a result, the number of possible states is not a problem anymore. However, if we use basic Value-Learning or Q-Learning, the performance wouldn’t be great if not enough states are explored and not enough times a state is explored. Thus, for this homework, we use an advanced method called “Approximate Q-Learning”. It inherits the advantages of Q-Learning, which computes and stores Q-values instead of state values for convenience when constructing policy. Furthermore, it doesn’t even need to store Q-values but only one vector representing the weights for each feature of a Q-state. It is possible that this method can still construct an optimal policy because the features of a state is actually more important than the state itself when deciding what action to take. Imagine that, the pacman is one step away from the ghost, no matter where they are, the best action a pacman can take is “run away!!”. But the above situation happening in different positions would be treated as different states, which is not efficient. To see more details for formulas, see q3 section.

An extra rule for the score is added to this game, which is that 0.1 point is taken when every time unit passes. This encourages pacman to eat food other than just avoid ghost. This rule isn’t applied to ValueIterGame because of the state space size.

In the game state, the following information is included:

"pacman_pos": position of pacman <vector>

"ghost1_pos": position of ghost <vector>

"food_num": number of foods not eaten yet <int>

"tiles": condition of each grid <list<int>>

"score": game score in such state

(get -0.1 for each time unit passed, +1 for each food eaten, +100 for winning, and -200 for losing.)

But like mentioned above, the state itself is not important. Except the scores, you don't need to access the values in this dictionary, but pass it to *LearningGame*.features() to get the features.

The methods you can use:

1. *LearningGame*.end_game(state<dict> = None) -> end_type<int>

   @state: state to test if ended, if not given, use *LearningGame*.state

   @Return end_type: condition of the state, 0 for not ended, 1 for losing, and 2 for winning.

2. *LearningGame*.nextstate(agent<string>, param1 = None, param2 = None) -> nextstate<dict>

   When it comes to reinforcement learning, the agent actually plays the game and learns from the results. But sometimes you need to know the Q-state without really changing the game state. Therefore, we provide two ways to use one of the two @agents. One is really to change the game state of *LearningGame* itself, while another one is to modify the given state passed by @param2.

   For this class, there are two possible agents: "sim_pac" and "sim_ghost_best_Q".

   "sim_pac": @param1: action<vector>, @param2: "from" state<dict>

   If @param2 is not given, modify the state of *LearningGame* itself, otherwise use @param2 as "from" state.

   @Return the resulted state by moving the pacman on state["pacman_pos"] with the given action.

"sim_ghost_best_Q": NO PARAMS NEEDED

This should be called after "sim_pac", so that the state of *LearningGame* then is actually a Q-state.

@Return the resulted state by moving the ghost on Q-state["ghost1_pos"] with the best action it should take.

3. *LearningGame*.features(state<dict> = None, action<vector> = None) -> featurenames <list<string>> or features<dict>

   @Return the feature names we use if no argument is given, otherwise the features of Q-state (Q(@state, @action))

Run "python LearningGame.py -p key" to play with keyboard.

"python LearningGame.py -p rein" to learn and play with reinforcement learning

If you run the above with one more command-line argument "-c", you can let your agent challenge a non-noisy ghost. This time, it can beat it without effort.

"rein.mp4" shows what it should look like if your code is correct.

- The main ideas of the questions are already described in the above sections, so we only talk about the implementation details here.

Once a ValueIterGame is constructed, the agents involved are initialized, and at the same time, an IntelligentAgent will start planning or learning.

The main two functions you need to implement for these agents are: 1. planning or learning, 2. move according to their policy.

q1. Implement IntelligentAgents.MDPAgent and IntelligentAgents.ValueIterationAgent

1. Iterate over all possible states (self.game.allstates()), update state value by:

$$V_{k+1}(s) \leftarrow \max \sum_{s'} T(S,a,s')[R(s,a,s') + \gamma V_k(s')]$$

Check if the value converges (the same as last iteration), if not, repeat.

2. For a given state, check all Q(s, a) from it, choose the action that makes the most Q-value. (This can be done with results stored while planning, so when really playing, just act according to the policy.)

q2 Implement IntelligentAgents.PolicyIterationAgent

Most of the time, policy iteration should converge sooner than value iteration.

1. In the beginning, you have to provide a trivial policy (e.g. always go right. (but it should be valid)) or totally random policy as an initial π.

(policy evaluation) Iterate over all possible states, update state value by:

$$V^{\pi}(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^{\pi}(S')]$$

(policy extraction) Iterate over all states, update the policy by:

$$\pi^*(s) = \text{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(S')]$$

Check if the policy converges (the same as last iteration), if not, repeat.

2. Just act according to the policy.

q3. Implement IntelligentAgents.LearningAgent

and IntelligentAgents.ApproximateQLearningAgent

1. Take an action according to the policy or randomly (decided by ε), make the ghost take its action, and see the result. Update weights by:

$$\text{difference} = [r + \gamma \max Q(s', a')] - Q(s, a)]$$

$$w_i \leftarrow w_i + \alpha \ [\text{difference}] f_i(s, a)$$

A Q-value is obtained by:

$$Q(s,a) = w_1 g_1(s,a) + w_2 g_2(s,a) \cdots + w_n g_n(s,a)$$

Where $g_i(s, a)$ represents the *i*th feature of Q-state Q(s, a).

Just to remind you, the features can be obtained by self.game.features(state, action).

Repeat until the game is ended. (test this by self.game.end_game())

Initialize the game state for the next game play. (by self.game.initialize())

Repeat the above for num_episode times.

2. For a given state, check all Q(s, a) from it, choose the action that makes the most Q-value.

- One more thing to say, some methods take an argument called “valid”, @valid is actually a function corresponding to self.game.valid. If you need to use it in those methods, just call by valid(pos). If you need to call such methods, pass self.game.valid as the argument.

e.g.

```
def will_move(self, valid, state):
        # some code
        if (valid(state[“pacman_pos”])):
                # some other code
…
def getValue(self, state):
        # some code
        for a in self.valid_moves(state[“pacman_pos”], self.game.valid):
                # some other code
```

****

DO NOT MODIFY ANY FILE OTHER THAN IntelligentAgents.py

****

If you got any question about this homework,

send an email to fad11204@yahoo.com.tw

****

The homework is graded by:

- Code correctness: whether the pacman can always beat the ghost
- Code cleanness: e.g. comments, meaningful variable names, …

Good luck!