Artificial Intelligence
AIC_4301C
Lab 4
2022-2023

Code and a report of this Lab have to be submitted in Balckboard before 23/04/2023 23:00.

- Create a folder Lab4_AIC_4301C_names-of-the-group-members
- Download the project **reinforcement_AIC.zip** from Blackboard, unzip it.
- Positions are represented by (x, y) Cartesian coordinates and any arrays are indexed by [x][y], with 'north' being the direction of increasing y, etc. By default, most transitions will receive a reward of zero, you can change this with the living reward option (-r).

1 Value Iteration

1. Write a value iteration agent in the class **ValueIterationAgent** in **valueIterationAgents.py** python file. ValueIterationAgent takes an MDP and computes k-step estimates of the optimal state values V_k using value iteration algorithm:

   Value Iteration Algorithm

   (a) $\forall s \in S, V_0(s) = 0$

   (b) Repeat the following update rule for the specified number of iterations (use option -i to specify the number of iterations):

   $$\forall s \in S, V_{k+1}(s) = max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

   These quantities are all displayed in the GUI: values are numbers in squares and q-values are numbers in square quarters.

   The methods to implement are:

   (a) **runValueIteration()**: runs the value iteration algorithm self.iterations times

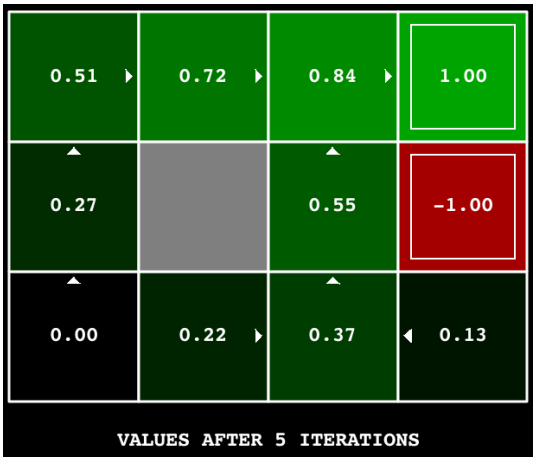
   (b) **computeActionFromValues(state)**: computes the best action according to the state value function (self.values[i] is the current state value of state i).

   (c) **computeq-valueFromValues(state, action)**: returns the q-value of the (state, action) pair given by the state value function given by self.values.

2. Test using the following command:

   python3 gridworld.py -a value -i 5

   You should have the following result with the values in the squares are state values and the policy is displayed in the GUI as arrows out from each square:



3. The following command computes a policy and executes it 10 times. The executions simply follows (this is not Q-learning) the computed policy using value iteration algorithm.

   python3 gridworld.py -a value -i 100 -k 10

   (a) Press a key to see q-values. q-values are displayed in the GUI as numbers in square quarters.

   (b) Press a key too see the 10 executions.

   You should find that the value of the start state (0,0) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

2 Q-Learning

Q-learning Algorithm

1. Compute a sample:
$$sample = R(s, a, s') + \gamma \max_{a'} Q(s', a')$$
2. Update $Q(s, a)$: $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha\ sample$

1. Write a **Q-learning** agent which learns by trial and error from interactions with the environment through its update(state, action, nextState, reward) method.

   You have to implement the following methods in the class **QLearningAgent** in the python file **qlearningAgents.py**. **Important: Make sure that in your computeValueFromQValues and computeActionFromQValues functions, you only access q-values by calling getQ-Value.** :

   (a) __init__(self, **args): in the init method create a dictionary to store the q-values of (state, action) already explored.

   (b) **update**: computes the new q-value using the Update rule of the Q-learning algorithm.

   (c) **computeValueFromQValues**: Returns maximum q-value Q(state,action) where the max is over legal actions. Note that if there are no legal actions, which is the case at the terminal state, you should return a value of 0.0.

   (d) **getQValue**: Returns the q-value Q(state,action). Returns 0.0 if we have never explored a state or the q-value otherwise.

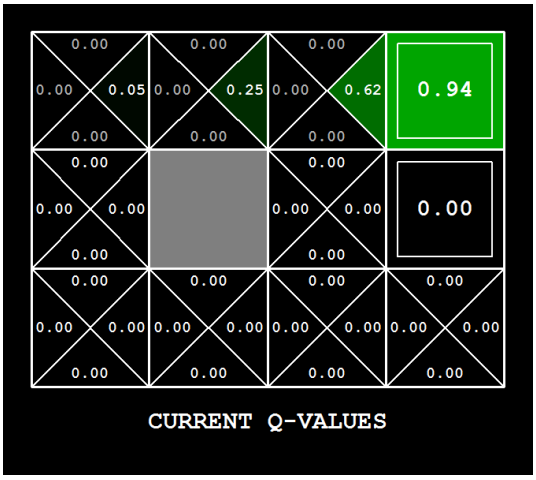
(e) **computeActionFromQValues**: Computes the best action to take in a state. Break ties randomly for better behavior using the random.choice() function. If there are no legal actions, which is the case at the terminal state, you should return None.

2. Test: Watch your Q-learner learn under manual control, using the keyboard:

   python3 gridworld.py -a q -k 5 -m

   The option -k control the number of episodes your agent gets to learn and -m is for manual control.

   To help with debugging, you can turn off noise by using the –noise 0.0 parameter. If you manually steer Pacman north and then east along the optimal path for four episodes, you should see the following q-values:



3. Complete your Q-learning agent by implementing epsilon-greedy action selection in the method getAction() in the class **QLearningAgent**.

   epsilon-greedy

   (a) Select n, a random uniform number in $[0, 1]$

   (b) If $n <$epsilon, act randomly (with probability epsilon)

   (c) If $n \geq$ epsilon, act on current policy (with probability 1-epsilon)

   epsilon-greedy exploration chooses random actions an epsilon fraction of the time, and follows its current best q-values otherwise. You can choose an element from a list uniformly at random by calling the random.choice function. You can simulate a binary variable with probability p of success by using util.flipCoin(p), which returns True with probability p and False with probability 1-p.

4. Test: Observe the following behavior of the agent in GridWorld (with epsilon = 0.3).

   python3 gridworld.py -a q -k 100

   Your final q-values should resemble those of your value iteration agent. However, your average returns will be lower than the q-values predict because of the random actions and the initial learning phase.

5. Test: With no additional code, you should be able to run a Q-learning crawler robot:

   python3 crawler.py

6. Test: Pacman Q-Learning

   PacmanQAgent is already defined for you in terms of the QLearningAgent you've already written. PacmanQAgent has default learning parameters epsilon=0.05, alpha=0.2, gamma=0.8. Without any code changes you should be able to run Q-learning Pacman for very tiny grids as follows:

   python3 pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid

   While a total of 2010 games will be played, the first 2000 games will not be displayed because of the option -x 2000, which designates the first 2000 games for training. Thus, you will only see Pacman play the last 10 of these games. During training, you will see results every 100 games with statistics

   The agent should win at least 80% of the time

3 Approximate Q-Learning

In approximate Q-learning:

- each q-state is represented as a feature vector $(f_1(s,a), f_2(s,a), \ldots)$
- $Q(s,a) = w_1 \times f_1(s,a) + w_2 \times f_2(s,a) + \ldots + w_n \times f_n(s,a)$

1. Approximate Q-learning agent learns weights for features of states.

   Approximate Q-Learning

   (a) $difference = [R(s,a,s') + \gamma \max_{a'} Q(s',a')] - Q(s,a)$

   (b) $w_i = w_i + \alpha \times difference \times f_i(s,a)$

   Features functions are provided in featureExtractors.py pyhthon file. Feature vectors are util.Counter (like a dictionary) objects containing the non-zero pairs of features and values and all omitted features have value zero. The keys in the dictionary define the identity of the feature.

   Write your implementation in ApproximateQAgent class in qlearningAgents.py. You have to write the following methods:

   (a) **getQValue**: returns $Q(state, action) = w_1 \times f_1(state, action) + w_2 \times f_2(state, action) + \ldots + w_n \times f_n(state, action)$ with n is the size of features = self.featExtractor.getFeatures(state, action)

   (b) **update**: update your weights based on Approximate Q-Learning algorithm

   (c) **final**: you might want to print your weights here for debugging

2. Test:

   python3 pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

   Your approximate Q-learning agent should win almost every time with these simple features, even with only 50 training games.