Report

2018/12/19

1 Introduction

In MDP, we solve a problem with given all the components such as the transition probabilities and rewards etc. But if we don't know that, we should use Model-Free learning in which the value functions will be figured out directly from the interactions with environment.

Model-Based Learning vs Model-Free Learning

Model-Based Learning learn the model empirically rather than values. In general, this approach will learn the optimal policy without evaluating a fixed policy. Model-Free are gonna the learn value functions directly from the environment. There are two methods:

1.Monte Carlo(MC)

2.Temporal-Difference(TD)

Monte Carlo method as a kind of Model-Free approaches learns directly from episodes of experience without bootstrapping.

Assume the episode has an end, the return is the total discounted reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T$$

The value function is the expected return:

$$v_{\pi(s)} = E_\pi \left[G_\pi | S_t = s \right]$$

2 Monte Carlo Implement

First of all, this program create the environment of the game in the main loop like

$$env = Env()$$

and the agent

$$agent = MCAgent(actions = list(range(env.n_actions)))$$

Then this program generate episodes and repeat it 1000 times.

Update Q-Function incrementally after each time the agent visit the state with the formula as given below.

$$V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t))$$

In the program:

```
state = str(reward[0])
if state not in visit_state:
    visit_state.append(state)
    G_t = self.discount_factor * (reward[1] + G_t)
    value = self.value_table[state]
    self.value_table[state] = (value +self.learning_rate * (G_t - value))
```

In this experiment, I add one more triangle with -1 reward at (4,3), just like Experiment(i), I change the program as follows

class Env(tk.Tk)
def _build_canvas(self):

$self.triangle13 = canvas.create\_image(350, 250, image = self.shapes[1])$

def step(self, action):
$elif next\_state == self.canvas.coords(self.triangle13):$
$reward = -1$
$done = True$

3 Experiment

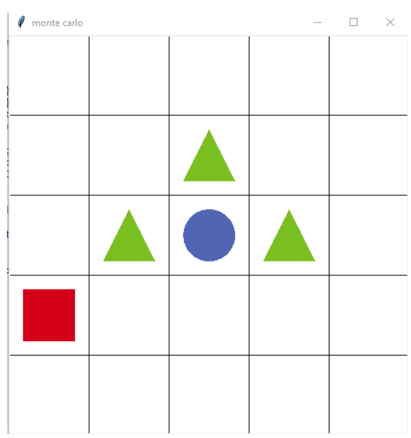


Figure 1:

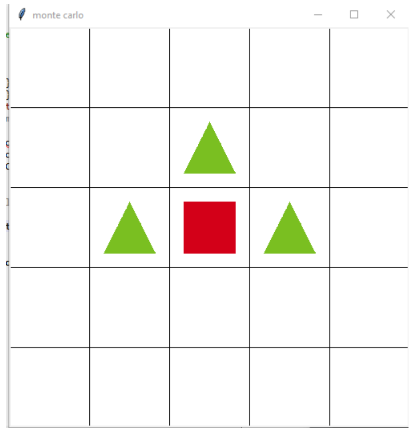
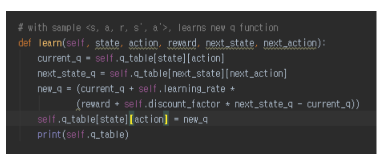


Figure 2:

4 SARSA Implement

SARSA is an on-policy algorithm, whose agent interacts with the environment and updates the policy based on actions taken. Lines 9-14 are initializes our variables, include actions, learning_rate, discount_factor, epsilon(i.e. for the epsilon-greedy approach), q_table. On line 17, the learn(self, state, action, reward, next_state, next_action) function updates the Q-table using the following equation:

$$Q\left(S,A\right)\leftarrow Q\left(S,A\right)+\alpha\left(R+\gamma Q\left(S',A'\right)-Q\left(S,A\right)\right)$$



```
# with sample <s, a, r, s', a'>, learns new q function
def learn(self, state, action, reward, next_state, next_action):
    current_q = self.q_table[state][action]
    next_state_q = self.q_table[next_state][next_action]
    new_q = (current_q + self.learning_rate *
            (reward + self.discount_factor * next_state_q - current_q))
    self.q_table[state][action] = new_q
    print(self.q_table)
```

Figure 3:

On line 27, an action is chosen for the next state using the get_action(self,state) function. The action chosen by it is done using the epsilon-greedy approach. See the lines 28-34, we randomly generate a number between 0 and 1 and see if it's smaller than epsilon. If it is smaller, then a random action is chosen using np.random.choice(self.actions) whereas we choose the action having the maximum value in the Q-table for state_action. Then we have:(State, Action, Reward, State', Action').

In this experiment, i add one more triangle with -1 reward at (4,3), i change the program as follows

class Env(tk.Tk)
def _build_canvas(self):

$self.triangle13 = canvas.create\_image(350, 250, image = self.shapes[1])$

def step(self, action):
$elif next\_state == self.canvas.coords(self.triangle13):$
$reward = -1$
$done = True$

5 Experiment

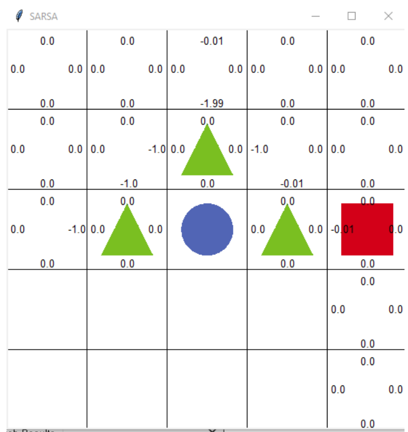
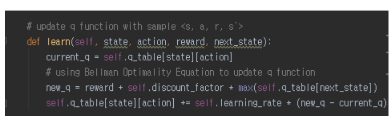


Figure 4:

6 Q-Learning Implement

Unlike SARSA, Q-Learning algorithm is an off-line TD control algorithm which update the learned action-value function, Q, directly approximates q*, the optimal action-value function, independent of the policy being followed. On line 16, the quite difference between SARSA and Q-Learning is function learn(self, state, action,reward, next_state) here by the following equation:

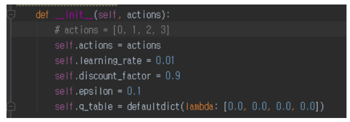
$$Q(s_t, a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q(s_{t+1}, a - Q(s_t, a_t))\right)$$



```
# update q function with sample <s, a, r, s'>
def learn(self, state, action, reward, next_state):
    current_q = self.q_table[state][action]
    # using Bellman Optimality Equation to update q function
    new_q = reward + self.discount_factor * max(self.q_table[next_state])
    self.q_table[state][action] += self.learning_rate * (new_q - current_q)
```

Figure 5:

Lines 7-13 initializes our variables.



```
def __init__(self, actions):
    # actions = [0, 1, 2, 3]
    self.actions = actions
    self.learning_rate = 0.01
    self.discount_factor = 0.9
    self.epsilon = 0.1
    self.q_table = defaultdict(lambda: [0.0, 0.0, 0.0, 0.0])
```

Figure 6:

On line 48, we create the Environment and the agent. On line 51, we start running the episodes. On line 52, the variable state stores the initial state using env.reset(). On line 24, the action is chosen using the epsilon-greedy approach. The lines 25-31, we randomly generate a number between 0 and 1 and see if it's smaller than epsilon. If it's smaller, then a random action is chosen using

$$action = np.random.choice(self.actions)$$

and if it's greater then we choose the action having the maximum value in the Q-table.

$$state\_action = self.q\_table[state]$$

$$action = self.arg\_max(state\_action)$$

In this experiment, the setting what I should do is the same as Experiment(i), I add one more triangle with -1 reward at (4,3), i change the program as follows

class Env(tk.Tk)
def _build_canvas(self):

$$self.triangle13 = canvas.create\_image(350, 250, image = self.shapes[1])$$

def step(self, action):
$elif next\_state == self.canvas.coords(self.triangle13):$
$reward = -1$
$done = True$

7 Experiment

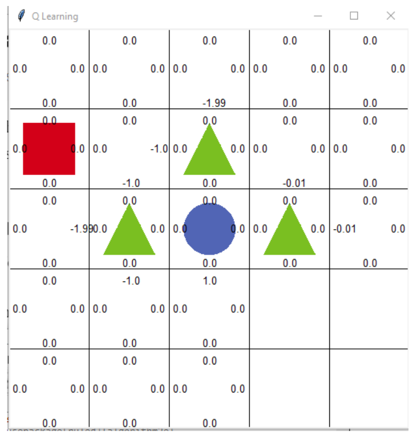


Figure 7: