02_gridworld_q_learning

September 29, 2021

1 Q-Learning in the GridWorld environment

Q-learning was an early RL breakthrough when it was developed by Chris Watkins for his PhD thesis in 1989. It introduces incremental dynamic programming to control an MDP without knowing or modeling the transition and reward matrices that we used for value and policy iteration in the previous section. A convergence proof followed three years later by Watkins and Dayan.

Q-learning directly optimizes the action-value function, q, to approximate q*. The learning proceeds off-policy, that is, the algorithm does not need to select actions based on the policy that's implied by the value function alone. However, convergence requires that all state-action pairs continue to be updated throughout the training process. A straightforward way to ensure this is by using an -greedy policy.

The Q-learning algorithm keeps improving a state-action value function after random initialization for a given number of episodes. At each time step, it chooses an action based on an -greedy policy, and uses a learning rate, , to update the value function, as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t_1} + \gamma \max_a Q(S_{t+1}, , a) - Q(S_t, A_t) \right]$$

Note that the algorithm does not compute expected values because it does know the transition probabilities. It learns the Q function from the rewards produced by the -greedy policy and its current estimate of the value function for the next state.

The use of the estimated value function to improve this estimate is called bootstrapping. The Q-learning algorithm is part of the TD learning algorithms. TD learning does not wait until receiving the final reward for an episode. Instead, it updates its estimates using the values of intermediate states that are closer to the final reward. In this case, the intermediate state is one time step ahead.

The notebook demonstrates how to build a Q-learning agent using the 3 x 4 grid of states from the Dynamic Programmimg example.

1.1 Imports & Settings

[1]:
```
%matplotlib inline

from pathlib import Path
from time import process_time
import numpy as np
import pandas as pd
```

```
from mdptoolbox import mdp
from itertools import product
```

1.2 Set up Gridworld

We first create our small gridworld as in the Dynamic Programmimg example.

1.2.1 States, Actions and Rewards

[2]:
```
grid_size = (3, 4)
blocked_cell = (1, 1)
baseline_reward = -0.02
absorbing_cells = {(0, 3): 1, (1, 3): -1}
```

[3]:
```
actions = ['L', 'U', 'R', 'D']
num_actions = len(actions)
probs = [.1, .8, .1, 0]
```

[4]:
```
to_1d = lambda x: np.ravel_multi_index(x, grid_size)
to_2d = lambda x: np.unravel_index(x, grid_size)
```

[5]:
```
num_states = np.product(grid_size)
cells = list(np.ndindex(grid_size))
states = list(range(len(cells)))
```

[6]:
```
cell_state = dict(zip(cells, states))
state_cell= dict(zip(states, cells))
```

[7]:
```
absorbing_states = {to_1d(s):r for s, r in absorbing_cells.items()}
blocked_state = to_1d(blocked_cell)
```

[8]:
```
state_rewards = np.full(num_states, baseline_reward)
state_rewards[blocked_state] = 0
for state, reward in absorbing_states.items():
    state_rewards[state] = reward
```

[9]:
```
action_outcomes = {}
for i, action in enumerate(actions):
    probs_ = dict(zip([actions[j % 4] for j in range(i, num_actions + i)],␣
 ↪probs))
    action_outcomes[actions[(i + 1) % 4]] = probs_
```

[10]:
```
action_outcomes
```

[10]:
```
{'U': {'L': 0.1, 'U': 0.8, 'R': 0.1, 'D': 0},
 'R': {'U': 0.1, 'R': 0.8, 'D': 0.1, 'L': 0},
 'D': {'R': 0.1, 'D': 0.8, 'L': 0.1, 'U': 0},
```

```
'L': {'D': 0.1, 'L': 0.8, 'U': 0.1, 'R': 0}}
```

1.2.2 Transition Matrix

[11]:
```
def get_new_cell(state, move):
    cell = to_2d(state)
    if actions[move] == 'U':
        return cell[0] - 1, cell[1]
    elif actions[move] == 'D':
        return cell[0] + 1, cell[1]
    elif actions[move] == 'R':
        return cell[0], cell[1] + 1
    elif actions[move] == 'L':
        return cell[0], cell[1] - 1
```

[12]:
```
state_rewards
```

[12]:
```
array([-0.02, -0.02, -0.02,  1.  , -0.02,  0.  , -0.02, -1.  , -0.02,
       -0.02, -0.02, -0.02])
```

[13]:
```
def update_transitions_and_rewards(state, action, outcome):
    if state in absorbing_states.keys() or state == blocked_state:
        transitions[action, state, state] = 1
    else:
        new_cell = get_new_cell(state, outcome)
        p = action_outcomes[actions[action]][actions[outcome]]
        if new_cell not in cells or new_cell == blocked_cell:
            transitions[action, state, state] += p
            rewards[action, state, state] = baseline_reward
        else:
            new_state= to_1d(new_cell)
            transitions[action, state, new_state] = p
            rewards[action, state, new_state] = state_rewards[new_state]
```

[14]:
```
rewards = np.zeros(shape=(num_actions, num_states, num_states))
transitions = np.zeros((num_actions, num_states, num_states))
actions_ = list(range(num_actions))
for action, outcome, state in product(actions_, actions_, states):
    update_transitions_and_rewards(state, action, outcome)
```

[15]:
```
rewards.shape, transitions.shape
```

[15]:
```
((4, 12, 12), (4, 12, 12))
```

1.3 Q-Learning

We will train the agent for 2,500 episodes, use a learning rate of = 0.1, and an =0.05 for the -greedy policy

[16]:
```
max_episodes = 2500
alpha = .1
epsilon = .05
gamma = .99
```

Then, we will randomly initialize the state-action value function as a NumPy array with the dimensions of [number of states and number of actions]:

[17]:
```
Q = np.random.rand(num_states, num_actions)
skip_states = list(absorbing_states.keys())+[blocked_state]
Q[skip_states] = 0
```

The algorithm generates 2,500 episodes that start at a random location and proceed according to the -greedy policy until termination, updating the value function according to the Q-learning rule:

[18]:
```
start = process_time()
for episode in range(max_episodes):
    state = np.random.choice([s for s in states if s not in skip_states])
    while not state in absorbing_states.keys():
        if np.random.rand() < epsilon:
            action = np.random.choice(num_actions)
        else:
            action = np.argmax(Q[state])
        next_state = np.random.choice(states, p=transitions[action, state])
        reward = rewards[action, state, next_state]
        Q[state, action] += alpha * (reward + gamma * np.
 ↪max(Q[next_state])-Q[state, action])
        state = next_state
process_time() - start
```

[18]:
```
0.67793093
```

The episodes take 0.6 seconds and converge to a value function fairly close to the result of the value iteration example from the previous section.

[19]:
```
pd.DataFrame(np.argmax(Q, 1).reshape(grid_size)).
 ↪replace(dict(enumerate(actions)))
```

[19]:

	0	1	2	3
0	R	R	R	L
1	U	L	L	L
2	U	L	L	D

[20]:
```
pd.DataFrame(np.max(Q, 1).reshape(grid_size))
```

[20]:

	0	1	2	3
0	0.877331	0.898778	0.945285	0.000000
1	0.845945	0.000000	0.706657	0.000000
2	0.807957	0.775816	0.745815	0.574385

1.4 PyMDPToolbox

1.4.1 Q Learning

[21]:
```
start = process_time()
ql = mdp.QLearning(transitions=transitions,
                   reward=rewards,
                   discount=gamma,
                   n_iter=int(1e6))

ql.run()
f'Time: {process_time()-start:.4f}'
```

[21]: 'Time: 10.8511'

[22]:
```
policy = np.asarray([actions[i] for i in ql.policy])
pd.DataFrame(policy.reshape(grid_size))
```

[22]:

	0	1	2	3
0	R	R	R	L
1	U	L	U	L
2	R	R	U	D

[23]:
```
value = np.asarray(ql.V).reshape(grid_size)
pd.DataFrame(value)
```

[23]:

	0	1	2	3
0	0.752821	0.913639	0.963127	0.00000
1	0.464707	0.000000	0.710295	0.00000
2	0.227155	0.501195	0.603085	0.23838