01_gridworld_dynamic_programming

September 29, 2021

1 Dynamic programming: Value and Policy Iteration

In this section, we will apply value and policy iteration to a toy environment that consists of a 3 x 4 grid that's depicted in the following diagram with the following features:

- **States**: 11 states represented as two-dimensional coordinates. One field is not accessible and the top two states in the rightmost column are terminal, that is, they end the episode.
- **Actions**: Movements on each step, that is, up, down, left, and right. The environment is randomized so that actions can have unintended outcomes. For each action, there is an 80% probability to move to the expected state, and 10% probability to move in an adjacent direction (for example, right or left instead of up or up or down instead of right).
- **Rewards**: As depicted in the right-hand side panel, each state results in -.02, except for the +1/-1 rewards in the terminal states:

The right panel of the preceding GridWorld diagram shows the optimal value estimate that's produced by Value Iteration and the corresponding greedy policy. The negative rewards, combined with the uncertainty in the environment, produce an optimal policy that involves moving away from the negative terminal state.

The results are sensitive to both the rewards and the discount factor. The cost of the negative state affects the policy in the surrounding fields, and you should modify the example in the corresponding notebook to identify threshold levels that alter the optimal action selection.

1.1 Imports & Settings

[1]:
```
%matplotlib inline

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

1.2.1 States, Actions and Rewards

We will begin by defining the environment parameters:

[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]
```

We will frequently need to convert between one-dimensional and two-dimensional representations, so we will define two helper functions for this purpose; states are one-dimensional and cells are the corresponding two-dimensional coordinates:

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

Furthermore, we will precompute some data points to make the code more concise:

[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)
```

We store the rewards for each state:

[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_
```

To account for the probabilistic environment, we also need to compute the probability distribution over the actual move for a given action:

[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}}
```

Now, we are ready to compute the transition matrix, which is the key input to the MDP.

1.2.2 Transition Matrix

The transition matrix defines the probability to end up in a certain state, S, for each previous state and action, A, $P(s' \mid s, a)$. We will demonstrate `pymdptoolbox`, and use one of the formats that's available to us to specify transitions and rewards. For both transition probabilities, we will create a `NumPy` array with dimensions of $A \times S \times S$.

First, we compute the target cell for each starting cell and move:

[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])
```

The following function uses the argument's starting `state`, `action`, and `outcome` to fill in the transition probabilities and rewards:

[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]
```

We generate the transition and reward values by creating placeholder data structures and iterating over the Cartesian product of $A \times S \times S$, as follows:

[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 PyMDPToolbox

We can also solve MDPs using the pymdptoolbox Python library, which includes a few more algorithms, including Q-learning.

1.3.1 Value Iteration

[16]:
```
gamma = .99
epsilon = 1e-5
```

To run `ValueIteration`, just instantiate the corresponding object with the desired configuration options and the rewards and transition matrices before calling the `.run()` method:

[17]:
```
vi = mdp.ValueIteration(transitions=transitions,
                        reward=rewards,
                        discount=gamma,
                        epsilon=epsilon)

vi.run()
f'# Iterations: {vi.iter:,d} | Time: {vi.time:.4f}'
```

[17]:
```
'# Iterations: 31 | Time: 0.0006'
```

[18]:
```
policy = np.asarray([actions[i] for i in vi.policy])
pd.DataFrame(policy.reshape(grid_size))
```

[18]:

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

[19]:
```
value = np.asarray(vi.V).reshape(grid_size)
pd.DataFrame(value)
```

[19]:

	0	1	2	3
0	0.884143	0.925054	0.961986	0.000000
1	0.848181	0.000000	0.714643	0.000000
2	0.808345	0.773328	0.736099	0.516083

1.3.2 Policy Iteration

The `PolicyIteration` function works similarly:

[20]:
```
pi = mdp.PolicyIteration(transitions=transitions,
                         reward=rewards,
                         discount=gamma,
                         max_iter=1000)

pi.run()
f'# Iterations: {pi.iter:,d} | Time: {pi.time:.4f}'
```

[20]:
```
'# Iterations: 7 | Time: 0.0560'
```

It also yields the same policy, but the value function varies by run and does not need to achieve the optimal value before the policy converges.

[21]:
```
policy = np.asarray([actions[i] for i in pi.policy])
pd.DataFrame(policy.reshape(grid_size))
```

[21]:

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

[22]:
```
value = np.asarray(pi.V).reshape(grid_size)
pd.DataFrame(value)
```

[22]:

	0	1	2	3
0	0.884143	0.925054	0.961986	-1.389785e-16
1	0.848181	0.000000	0.714643	5.749281e-16
2	0.808345	0.773328	0.736099	5.160828e-01

1.4 Value Iteration

[23]:
```
skip_states = list(absorbing_states.keys())+[blocked_state]
states_to_update = [s for s in states if s not in skip_states]
```

Then, we initialize the value function and set the discount factor gamma and the convergence threshold epsilon:

[24]:
```
V = np.random.rand(num_states)
V[skip_states] = 0
```

[25]:
```
gamma = .99
epsilon = 1e-5
```

The algorithm updates the value function using the Bellman optimality equation, and terminates when the L1 norm of V changes less than epsilon in absolute terms:

[26]:
```
iterations = 0
start = process_time()
converged = False
while not converged:
    V_ = np.copy(V)
    for state in states_to_update:
        q_sa = np.sum(transitions[:, state] * (rewards[:, state] + gamma* V),␣
 ↪axis=1)
        V[state] = np.max(q_sa)
    if np.sum(np.fabs(V - V_)) < epsilon:
        converged = True

    iterations += 1
    if iterations % 1000 == 0:
        print(np.sum(np.fabs(V - V_)))

f'# Iterations {iterations} | Time {process_time() - start:.4f}'
```

[26]:
```
'# Iterations 24 | Time 0.0023'
```

1.4.1 Value Function

[27]:
```
print(pd.DataFrame(V.reshape(grid_size)))
```

```
          0         1         2         3
0  0.884143  0.925054  0.961986  0.000000
1  0.848181  0.000000  0.714643  0.000000
2  0.808345  0.773328  0.736099  0.516083
```

[28]:
```
np.allclose(V.reshape(grid_size), np.asarray(vi.V).reshape(grid_size))
```

[28]:
```
True
```

1.4.2 Optimal Policy

[29]:
```
for state, reward in absorbing_states.items():
    V[state] = reward

policy = np.argmax(np.sum(transitions * V, 2),0)
policy
```

[29]:
```
array([2, 2, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0])
```

[30]:
```
pd.DataFrame(policy.reshape(grid_size)).replace(dict(enumerate(actions)))
```

[30]:

	0	1	2	3
0	R	R	R	L

```
1  U  L  L  L
2  U  L  L  L
```

1.5 Policy Iteration

Policy iterations involves separate evaluation and improvement steps. We define the improvement part by selecting the action that maximizes the sum of expected reward and next-state value. Note that we temporarily fill in the rewards for the terminal states to avoid ignoring actions that would lead us there:

[31]:
```
def policy_improvement(value, transitions):
    for state, reward in absorbing_states.items():
        value[state] = reward
    return np.argmax(np.sum(transitions * value, 2),0)
```

[32]:
```
V = np.random.rand(num_states)
V[skip_states] = 0
pi = np.random.choice(list(range(num_actions)), size=num_states)
```

The algorithm alternates between policy evaluation for a greedily selected action and policy improvement until the policy stabilizes:

[33]:
```
iterations = 0
start = process_time()
converged = False
while not converged:
    pi_ = np.copy(pi)
    for state in states_to_update:
        action = policy[state]
        V[state] = np.dot(transitions[action, state], (rewards[action, state] +␣
 ↪gamma* V))
        pi = policy_improvement(V.copy(), transitions)
    if np.array_equal(pi_, pi):
        converged = True
    iterations += 1

f'# Iterations {iterations} | Time {process_time() - start:.4f}'
```

[33]:
```
'# Iterations 3 | Time 0.0009'
```

Policy iteration converges after only three iterations. The policy stabilizes before the algorithm finds the optimal value function, and the optimal policy differs slightly, most notably by suggesting up instead of the safer left for the field next to the negative terminal state. This can be avoided by tightening the convergence criteria, for example, by requiring a stable policy of several rounds or adding a threshold for the value function.

[34]:
```
pd.DataFrame(pi.reshape(grid_size)).replace(dict(enumerate(actions)))
```

[34]:

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

[35]:
```
pd.DataFrame(V.reshape(grid_size))
```

[35]:

	0	1	2	3
0	0.756333	0.882232	0.933790	0.000000
1	0.683594	0.000000	0.480169	0.000000
2	0.612364	0.552599	0.506767	0.307299