Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

Q-LEARNING

Karol Działowski

nr albumu: 39259
przedmiot: Uczenie ze wzmocnieniem

Szczecin, 18 stycznia 2021

Spis treści

1 Cel laboratorium 1

2 Zadanie 1 - zapałki 1

3 Zadanie 3 - dwuwymiarowa siatka 4

4 Wnioski 8

1 Cel laboratorium

Celem laboratorium nr 3 była implementacja algorytmu q-learning w celu wyznaczenia sub-optymalnej strategii przy eksploracji środowiska.

2 Zadanie 1 - zapałki

Przedmiotem rozważań jest gra w zapałki. Gracze mają do dyspozycji pewną liczbę (np. $N = 10$) zapałek. W grze bierze udział dwóch graczy, którzy na przemian zabierają 1, 2 lub 3 zapałki. Przegrywa ten gracz, który zabiera zapałki jako ostatni.

Należało wyznaczyć stratgię z wykorzystaniem algorytmu Q-Learn.

Kod źródłowy 1: Zadanie 1 - zapałki
Źródło: Opracowanie własne

```
1 import random
```

```python
import numpy as np

def take_action(state, action):
    # if losing state
    if state == 1:
        return 0, 0, True

    state_next = state - (action+1)
    # if winning move return R and state
    if state_next == 1:
        return 0, 1, True

    # random opponents action
    action_opponent = random_action()

    # calculate reward
    R = 0
    done = False
    state_next = state_next - (action_opponent+1)

    if state_next <= 0:
        X_n = 0
        R = 1
        done = True

    return state_next, R, done

def random_action():
    return random.choice([0, 1])

def main():
    N = 10  # starting state
    q = np.random.rand(N+1, 2)

    gamma = 0.01
    epochs = 5000
    epsilon = 0.2  # exploration

    for i in range(epochs):
        state = N
        done = False

        while not done:
            print(f"Epoka {i}/{epochs}")

            # chose action
            if random.random() < epsilon:
                action = random_action()
            else:
                action = q[state].argmax()

```

```
            # take action
            state_next, reward, done = take_action(state, action)

            # update delta
            q[state][action] = reward + gamma * q[state_next].max()
            state = state_next

    print()
    print("Macierz Q")
    print(q)

    print("Wyznaczona strategia")
    policy = np.argmax(q, axis=1)+1
    print(policy)

    print("Sprawdzenie strategii")
    print(policy[9] == 2 and
            policy[8] == 1 and
            policy[6] == 2 and
            policy[5] == 1 and
            policy[4] == 1 and
            policy[3] == 2 and
            policy[2] == 1)


if __name__ == "__main__":
    main()
```

Wyznaczona strategia pokrywa się z strategią optymalną wyznaczoną na poprzednich laboratoriach.

Kod źródłowy 2: Zadanie 1 - wyznaczona strategia
Źródło: Opracowanie własne

```
Macierz Q
[[9.89671802e-01 6.06568403e-01]
 [9.89671802e-03 9.89671802e-03]
 [1.00989672e+00 1.00861157e+00]
 [9.89671802e-05 1.00989672e+00]
 [1.00989672e-02 9.89671802e-05]
 [1.00989672e-02 1.00989672e-02]
 [9.89671802e-07 1.00989672e-02]
 [1.00989672e-04 9.89671802e-07]
 [1.00989672e-04 1.00989672e-04]
 [2.06204220e-01 8.61157285e-01]
 [1.00989672e-06 9.89671802e-09]]
Wyznaczona strategia
[1 1 1 2 1 1 2 1 1 2 1]
Sprawdzenie strategii
True
```

3 Zadanie 3 - dwuwymiarowa siatka

Zaimpelementowano symulator środowiska o siatce 10×10 z przeszkodami. Ustawiono stany absorbujące. Wykorzystano algorytm q-learn do nauczenia strategii.

Kod źródłowy 3: Zadanie 2 - grid
Źródło: Opracowanie własne

```
import random
import numpy as np


class Grid:
    def __init__(self):
        self.height = 10
        self.width = 10
        self.actions = [0, 1, 2, 3]
        self.state_count = self.height * self.width
        self.action_count = len(self.actions)

        self.policy_print_characters = {0: "<", 1: ">", 2: "^", 3: "v"}

        self.grid = np.zeros((self.height, self.width))
        # end points
        self.grid[0, 0] = 10
        self.grid[-1, 0] = 50

        # obstacles
        self.grid[3, 2:8] = -1

        self.grid[2:8, 6] = -1
        self.grid[2:8, 5] = -1
        self.grid[2:8, 4] = -1
        self.grid[2:8, 3] = -1


    def random_position(self):
        """Find random starting position"""
        while True:
            h = random.randrange(0, self.height)
            w = random.randrange(0, self.width)
            if self.grid[h, w] == 0:
                return (h, w)

    def is_terminal(self, state):
        """Terminal states are those with reward greater than 0"""
        if self.grid[state[0], state[1]] > 0:
            return True
        else:
            return False

```

```
    def to_string(self, state=None, policy=None):
        res = " " + " " * self.width + " " + "\n"
        for i in range(self.height):
            res += " "
            for j in range(self.width):
                if self.grid[i, j] == -1:
                    res += "#"
                elif self.grid[i, j] > 0:
                    res += "X"
                elif state is not None and state[0] == i and state[1] == j:
                    res += "¤"
                elif policy is not None:
                    res += self.policy_print_characters[policy[i, j]]

                else:
                    res += " "
            res += " "
            res += "\n"
        res += " " + " " * self.width + " "
        return res

    def __str__(self):
        return self.to_string()

    def available_actions(self, state):
        actions = []
        if self.is_terminal(state):
            return actions

        h, w = state
        if h > 0:
            if self.grid[h-1, w] >= 0:
                actions.append(2)  # up
        if w > 0:
            if self.grid[h, w-1] >= 0:
                actions.append(0)  # left
        if h < self.height-1:
            if self.grid[h+1, w] >= 0:
                actions.append(3)  # down
        if w < self.width-1:
            if self.grid[h, w+1] >= 0:
                actions.append(1)  # right

        return actions


    def take_action(self, state, action):
        h, w = state
        state_next = [h, w]

        if action == 0:  # left
```

```
            state_next[1] = w-1
        if action == 1:  # right
            state_next[1] = w+1
        if action == 2:  # up
            state_next[0] = h-1
        if action == 3:  # down
            state_next[0] = h+1

        done = self.is_terminal(state_next) or self.available_actions(state_next) ==
[]
        reward = self.grid[state_next[0], state_next[1]]
        return state_next, reward, done



def main():
    import numpy as np
    import time
    import os

    env = Grid()

    qtable = np.random.rand(env.height, env.width, env.action_count)

    epochs = 5000
    gamma = 0.05
    epsilon = 0.1

    for i in range(epochs):
        state = env.random_position()
        reward = 0
        done = False
        steps = 0


        while not done:
            h, w = state

            # count steps to finish game
            steps += 1

            # exploration
            if np.random.random() < epsilon:
                action = np.random.choice(env.available_actions(state))
            # best action
            else:
                action = qtable[h, w].argmax()
                available_actions = env.available_actions(state)
                if action not in available_actions:
                    action = np.random.choice(env.available_actions(state))

```

```
145
146             # take action
147             state_next, reward, done = env.take_action(state, action)
148
149             # update qtable value with Bellman equation
150             qtable[h, w, action] = reward + gamma * np.max(qtable[state_next])
151
152             # update state
153             # print("Action", action)
154             # print("Step", steps)
155             state = state_next
156
157             #print(env.to_string(state=state))
158
159         #print("Epoka: ", i + 1, "kroki:", steps)
160         if i % 100 == 0:
161             print("Wyznaczona strategia")
162             policy = np.argmax(qtable, axis=2)
163             print(env.to_string(policy=policy))
164
165     print("Wyznaczona strategia")
166     policy = np.argmax(qtable, axis=2)
167     print(policy)
168     print(env.to_string(policy=policy))
169
170 if __name__ == "__main__":
171     main()
```

Wyznaczona strategia w 5000 epokach przy $\gamma = 0.9$, $\epsilon = 0.2$:

Kod źródłowy 4: Zadanie 2 - grid
Źródło: Opracowanie własne

```
1  Wyznaczona strategia
2  [[0 0 0 0 3 0 3 1 1 3]
3   [2 0 0 2 0 0 0 1 1 2]
4   [1 0 0 0 2 2 1 1 1 2]
5   [2 2 3 0 0 0 0 1 1 2]
6   [1 2 0 0 0 0 1 1 1 2]
7   [1 2 0 0 0 0 1 1 2 2]
8   [1 0 0 0 0 0 1 1 1 3]
9   [3 3 3 0 3 3 1 3 3 3]
10  [3 3 1 3 3 0 3 0 0 2]
11  [0 0 0 0 0 0 0 1 2 2]]
12
13 X<<<v<v>> v
14 ^<<^<<<>>^
15 ><<# ###>>^
16 ^^# #####>^
17 >^<# ###>>^
18 >^<# ###>^^
19 ><<####>> v
```

```
vvv#### vvv
vv>vv<v <<^
X <<<<<<>^^
```

4 Wnioski

Zaimplementowano algorytm Q-Learning na dwóch środowiskach - zapałki oraz siatka z przeszkodami.

Wyznaczona strategia dla zapałek pokrywa się z wyznaczoną strategią optymalną z poprzedniego zadania.

W przypadku problemu dla siatki algorytm nie wyznaczył optymalnej strategii przy danych parametrach. W okolicy stanów końcowych strategia jest dobra, ale im dalej od punktu tym więcej błędnych akcji jest ocenianych jako najlepsze. Prawdopodobnie wynika to ze złego doboru parametrów uczących.