ANSWERS

8.5 Learning by reward

Learning · Robot club · about 30 min

What this lesson is about

Q-learning: a table of how good each action is, filled in by trying.

- States and actions
- The table
- The reward, and the rule
- What it learns

Questions

8 marks in all

1. What makes learning by reward different from the classifiers earlier in the module? [1 mark]

- ☐ **A** There are no labels: the robot tries actions and is told afterwards how well they went
- ☐ **B** It needs more labelled samples
- ☐ **C** It uses the camera instead of the depth grid
- ☐ **D** It never makes mistakes while learning

> **Answer:** A. Nobody tells it the right action. A reward after each try is all it gets.

2. The states are near (under 20 cm), mid (under 40 cm) and far, and the actions are forward and turn. How many numbers does the Q table hold? [1 mark]

> **Answer:** 6. One number for every state and action: 3 states x 2 actions = 6.

3. What does this program print? [1 mark]

```
def state(ahead):
    return "near" if ahead < 20 else ("mid" if ahead < 40 else "far")

print(state(12))
print(state(20))
print(state(40))
```

> **Answer:**
>
> ```
> near
> mid
> far
> ```
>
> 12 is under 20, so near. 20 is not under 20 but is under 40, so mid. 40 is not under 40, so far.

4. What does this program print? [1 mark]

```
Q = {"near": {"forward": 0.0, "left": 0.0}, "far": {"forward": 2.0, "left": 0.5}}
s, a, s2 = "near", "forward", "far"
reward = 1.0
Q[s][a] += 0.3 * (reward + 0.8 * max(Q[s2].values()) - Q[s][a])
print(round(Q[s][a], 2))
```

Answer:

```
0.78
```

The surprise is 1.0 + 0.8 x 2.0 - 0 = 2.6, and the table moves 0.3 of the way: 0.3 x 2.6 = 0.78.

5. Why does the robot sometimes pick an action at chance instead of the best one it knows? [1 mark]

- ☐ **A** Those tries are how it discovers anything it does not already know
- ☐ **B** To save battery
- ☐ **C** Because the table is full
- ☐ **D** So it bumps into things on purpose

Answer: A. Always choosing the best known action means never finding a better one. The fraction, epsilon, shrinks as it learns.

6. What is `was_bumped` for in the reward loop? [1 mark]

- ☐ **A** `bumped()` stays true for a third of a second, so it makes one bump cost -20 only once
- ☐ **B** It counts how many times the robot has turned
- ☐ **C** It stops the program after the first bump
- ☐ **D** It remembers which side the bump was on

Answer: A. Without it, one touch could be punished on several steps in a row.

7. After learning, what does the table usually say in the `near` state? [1 mark]

- ☐ **A** Turn is worth more than forward
- ☐ **B** Forward is worth more than turn
- ☐ **C** Both actions are worth zero
- ☐ **D** Both actions are strongly negative

Answer: A. Forward from near leads to bumps, and turning leads to open space where forward pays. Nobody wrote that rule.

8. Why does a turn keep going the same way until the robot drives forward again? [1 mark]

- ☐ **A** A robot that picks left or right afresh every step can swap between them in a corner for ever
- ☐ **B** Turning left and right costs different amounts
- ☐ **C** The depth grid can only see one side at a time
- ☐ **D** It saves battery

Answer: A. In a corner, turning one way shows a nearer wall and turning back shows the first one again. Committing to a direction always gets it out.

The task: learn to wander

Let the robot learn for 88 seconds. It may bump early on. After 60 seconds it must not bump at all, and it must keep getting about the mat: rocking back and forth or spinning in one spot does not count. It must drive at least 250 cm in total. Print `bumps: <n>` at the end.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()
bumps = 0
# do this 880 times (tick counts from 0)
for tick in range(880):
    # drive forward at 60 (keeps going until the next command)
    forward(60)
    # pause 0.1 s (the robot keeps doing what it was told)
    wait(0.1)
    if bumped():
        bumps += 1
        # spin clockwise on the spot at 60
        turn_right(60)
        # pause 0.5 s (the robot keeps doing what it was told)
        wait(0.5)
# all motors off
stop()
print('bumps:', bumps)
```

The hint students can ask for: Let the robot learn by reward: bumping costs, ground covered pays. It may bump early on; after 60 seconds it must not bump at all and must keep getting about the mat, not stay in one spot, and it must have driven at least 250 cm. Print `bumps: <n>` at the end.

A solution

```
from bugbot import *
connect()
import random
random.seed(1)
ACTIONS = ["forward", "turn"]
STATES = ["near", "mid", "far"]
Q = {s: {a: 0.0 for a in ACTIONS} for s in STATES}     # the table: how good each action is
in each state

def state():
    level = tof_grid()[16:32]                            # the two level rows, all eight
columns
    ahead = min(level)                                   # the nearest thing anywhere in
front
    return "near" if ahead < 20 else ("mid" if ahead < 40 else "far")

def room_left():
    level = tof_grid()[16:32]
    return min(level[0:3] + level[8:11]) >= min(level[5:8] + level[13:16])

way = None                                               # the way the robot is turning, or
None
def act(a):
    global way
    if a == "forward":
```

```
        forward(60); wait(0.25)
        way = None
        return
    if way is None:                                     # a new turn: towards the side
with more room
        way = "left" if room_left() else "right"
    if way == "left":
        turn_left(100)
    else:
        turn_right(100)
    wait(0.2)

bumps = 0
was_bumped = False
epsilon = 0.3
while clock() < 88:
    s = state()
    if random.random() < epsilon:
        a = random.choice(ACTIONS)                      # explore
    else:
        a = max(Q[s], key=Q[s].get)                     # exploit: the best known action
    act(a)
    reward = 1.0 if a == "forward" else -0.2            # driving pays; turning costs a
little
    hit = bumped()
    if hit and not was_bumped:                          # a new bump
        reward = -20.0
        bumps += 1
    elif hit:                                           # still pushing against it
        reward = -5.0
    was_bumped = hit
    s2 = state()
    Q[s][a] += 0.3 * (reward + 0.8 * max(Q[s2].values()) - Q[s][a])   # the learning rule
    epsilon = epsilon * 0.985                           # explore less and less
stop()
print("bumps:", bumps)
for s in STATES:
    print(s, {a: round(v, 1) for a, v in Q[s].items()})
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.