ANSWERS

U12.4 Policy search on a robot

Learning, and the capstone · University · about 40 min

What this lesson is about

Hill climbing on a real machine: a noisy score, a fair trial, and a budget measured in seconds.

- The setting
- Hill climbing
- The trial is where the rigour is
- Exploration and exploitation
- Why not just write the controller?

Questions

7 marks in all

1. This is the lesson's hill climb on a score whose true best value is 23. What does it print? [1 mark]

```
def score(v):
    return abs(v - 23.0) + 2.0

value, step = 0.0, 20.0
best = score(value)
for i in range(8):
    trial = value + step
    s = score(trial)
    if s < best:
        value, best = trial, s
    else:
        step = -step * 0.6
print(round(value, 2), round(best, 2), round(step, 2))
```

Answer:

```
22.59 2.41 0.93
```

It keeps stepping while the score falls, and each time it overshoots it turns round with a step 0.6 times smaller, closing in on 23 without ever using a derivative.

2. Why is a derivative free search the right tool for tuning the lateral offset on the robot? [1 mark]

- ☐ **A** The score comes out of a physical trial, not a formula, so there is no derivative to use
- ☐ **B** Derivative free methods always find the global optimum
- ☐ **C** Derivatives are too slow to compute for one parameter
- ☐ **D** The offset has no effect on the score's slope

Answer: A. Hill climbing only compares scores. The price is that it finds a local optimum, which for one well behaved parameter is usually the global one.

3. Each trial starts wherever the previous one ended. What is wrong with the scores? [1 mark]

- ☐ **A** Each score measures the previous policy as much as the current one
- ☐ **B** They are noisier but still unbiased
- ☐ **C** Nothing, as long as the trials are the same length
- ☐ **D** The search will take fewer trials

> **Answer:** A. Every trial must start from the same state. The out and back design returns the robot to where it began.

4. A method needs 10,000 evaluations and each out and back trial takes 1.6 s of robot time. How many hours is that, to 2 decimal places? [1 mark]

> **Answer:** 4.44 (accept within 0.01). 10,000 x 1.6 = 16,000 s, and 16,000 / 3600 = 4.44 hours, before any resets or battery changes. That is why simulation exists.

5. The noise in one trial's score is about as large as the difference between two neighbouring offsets. What is the search doing? [1 mark]

- ☐ **A** Following coin flips; repeat and average the trials, or make them longer
- ☐ **B** Converging faster, because noise helps it explore
- ☐ **C** Finding the global optimum
- ☐ **D** Nothing different, because the shrinking step handles noise

> **Answer:** A. One trial cannot tell the two policies apart, so the comparisons are chance. Averaging costs budget, and you should know which you are spending.

6. Which statements about exploration and exploitation are right? [1 mark]

Tick every answer that is true.

- ☐ **A** Hill climbing explores with large early steps and exploits as the step shrinks
- ☐ **B** An upper confidence bound rule explores in proportion to how little an option has been tried
- ☐ **C** Exploiting too early polishes a mediocre policy
- ☐ **D** A good enough method makes the trade-off disappear
- ☐ **E** Epsilon-greedy never takes a random action once it has a best option

> **Answer:** A, B, C. The trade-off is a property of learning from your own actions and never goes away. Epsilon-greedy keeps a decaying fraction of random actions.

7. Feeding flow()[0] back into the lateral command would cancel the leak without learning. What does the learned offset offer that feedback does not? [1 mark]

- ☐ **A** It works with no sensor during the run and costs nothing at run time
- ☐ **B** It adapts to changes in the leak that happen after tuning
- ☐ **C** It removes the need for any trials
- ☐ **D** It handles situations the learner never saw

> **Answer:** A. Feedback handles changes the learner never saw. Real systems use both: learned feedforward for what is predictable, feedback for what is not.

The task: learn the offset that drives it straight

Hill climb the lateral offset using short out and back trials, plot `score` as you go, and then drive the robot at least 90 cm up into the green lane.

```
from bugbot import *
connect()

DT = 0.1
CMD = 60
```

The hint students can ask for: This robot slides sideways whenever it drives forward. The policy is one number, a lateral command held on all the time, and its score is how far the robot slid during a short trial, measured by flow(). Make each trial out and back so the robot stays put, hold the heading square so the trial measures sliding and nothing else, then hill climb: try a step, keep it if it scored better, turn round and shrink the step if it did not.

A solution

```
from bugbot import *
connect()

DT = 0.1
CMD = 60
SETTLE = 3                          # ticks before the reading counts: the motors take a
moment to come up

def turn_cmd():
    """Hold the robot square, so that sliding sideways is the only thing the trial
measures."""
    err = (imu()[0] + 180) % 360 - 180
    if abs(err) < 2.0:
        return 0
    cmd = max(18.0, min(30.0, abs(1.5 * err)))     # below 15 the drive does nothing at all
    return -cmd if err > 0 else cmd

def leg(fwd, lat, seconds):
    """Drive for a moment and return how far the robot slid sideways, from the flow
sensor."""
    slide = 0.0
    for i in range(SETTLE + int(seconds / DT)):
        drive(fwd, lat, turn_cmd())
        if i >= SETTLE:
            slide += flow()[0] * DT
        wait(DT)
    return slide

def score(offset):
    """One trial: out and back, so the robot ends where it started and the trial can be
repeated."""
    out = leg(CMD, offset, 0.8)
    back = leg(-CMD, -offset, 0.8)
    stop()
    wait(0.2)
    return abs(out) + abs(back)

offset, step = 0.0, 16.0
best = score(offset)
```

```
for i in range(11):
    trial = offset + step
    s = score(trial)
    if s < best:
        offset, best = trial, s
    else:
        step = -step * 0.6         # wrong way, and getting close: turn round and take a
smaller step
    plot("score", best)
    plot("offset", offset)
print("offset:", round(offset, 1))
print("score:", round(best, 2))

# the run itself, with the offset held on all the way
travelled = 0.0
for tick in range(300):
    drive(CMD, offset, turn_cmd())
    travelled += flow()[1] * DT
    if travelled > 105:
        break
    wait(DT)
stop()
wait(0.5)
print("travelled:", round(travelled, 1))
```

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