6ELEN018W - Applied Robotics
Lecture 5: Intelligent Control Algorithms, Part II (cont'd) - The Gymnasium RL Environment

Dr Dimitris C. Dracopoulos

Value Function

How can a robot calculate the optimum action at each state?

- What if each state s_i has a value associated with it, measuring the total all future reward received after starting from this state and following a policy of actions?
- Then the robot could choose an action that will lead to a state with the highest value.

For each possible policy π the robot might adopt, we can define an evaluation function over states

$$\begin{aligned} V^{\pi}(s) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \\ &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned}$$

where $r_t, r_{t+1}, \ldots$ are generated by following policy π starting at state s

Now, the task is to learn the optimal policy π^*:

$$\pi^*(s) = \arg\max_a [r(s,a) + \gamma V^*(\delta(s,a))]$$

Q Function - An Alternative to choose Robot Actions

Define new function very similar to V^*

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a))$$

If agent learns Q, it can choose optimal action even without knowing δ! (the function which describes the transition between the current state and the next one if the robot takes a specific action)

$$\pi^*(s) = \arg\max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

$$\pi^*(s) = \arg\max_a Q(s, a)$$

Q is the evaluation function the agent will learn

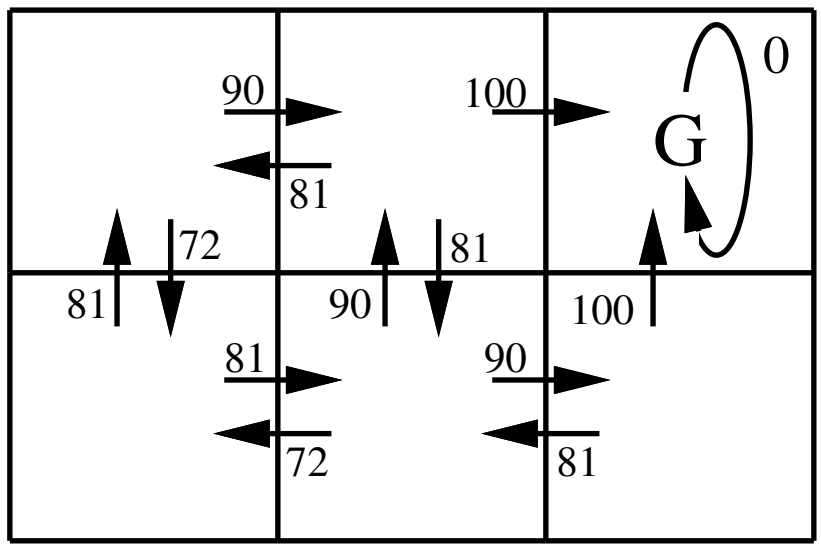


Figure 1: $Q(s, a)$ values for the grid problem previously seen.

Training Rule to Learn Q

Note Q and V^* closely related:

$$V^*(s) = \max_{a'} Q(s, a')$$

Which allows us to write Q recursively as

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))) \\ &= r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned}$$

Let $\hat{Q}$ denote learner's current approximation to Q. Consider training rule

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

where s' is the state resulting from applying action a in state s.

Q Learning Pseudocode for Deterministic Worlds

For each s, a initialise table entry $\hat{Q}(s, a) \longleftarrow 0$

Observe current state s

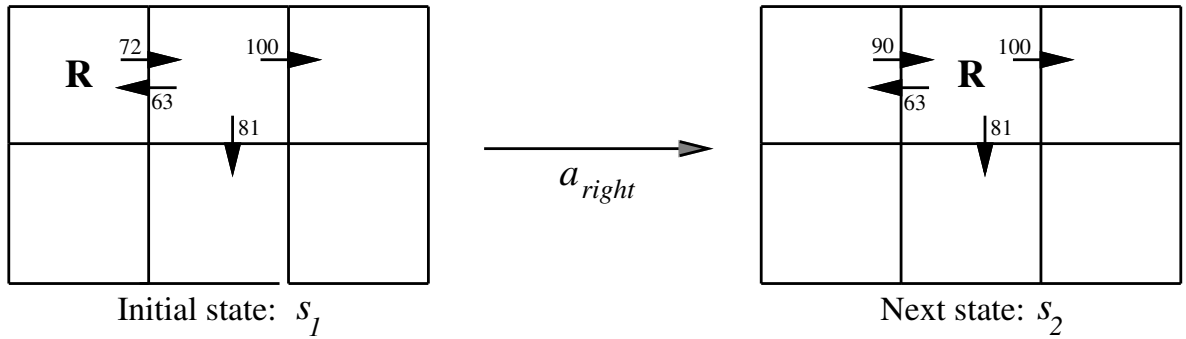
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \longleftarrow s'$

Updating $\hat{Q}$



$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \ \max\{63, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

notice if rewards non-negative, then

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a)$$

and

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s, a) \leq Q(s, a)$$

Nondeterministic Case

What if reward and next state are non-deterministic?

We redefine V, Q by taking expected values

$$\begin{aligned} V^{\pi}(s) &\equiv E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots] \\ &\equiv E[\sum_{i=0}^{\infty} \gamma^i r_{t+i}] \\ &\equiv E[r_t + \gamma V^{\pi}(s+1)] \end{aligned} \tag{1}$$

$$Q(s,a) \equiv E[r(s,a) + \gamma V^*(\delta(s,a))]$$

Nondeterministic Case

Q learning generalises to nondeterministic worlds

Alter training rule to

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \max_{a'} \hat{Q}_{n-1}(s',a')]$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}$$

Can still prove convergence of $\hat{Q}$ to Q.

Temporal Difference Learning

Q learning (TD(0) algorithm): reduce discrepancy between successive Q estimates

One step time difference:

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

Why not two steps?

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

Or n?

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

Blend all of these:

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda) \left[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \cdots \right]$$

Families of Reinforcement Learning Algorithms

1. *Dynamic Programming*: based on Bellman equation (1), well developed mathematically, but require a complete and accurate model of the environment.
2. *Monte Carlo Methods*: do not require a model but not appropriate for step-by-step incremental learning.
3. *Temporal Difference Methods* (e.g. Q-learning (which is the TD(0) algorithm), temporal difference learning): require no model, they are fully incremental, but are more complex to analyse.

Other Improvements on what was discussed

- Extend to continuous action, state: Replace $\hat{Q}$ table with neural net or other generaliser
- Learn and use $\hat{\delta} : S \times A \longrightarrow S$

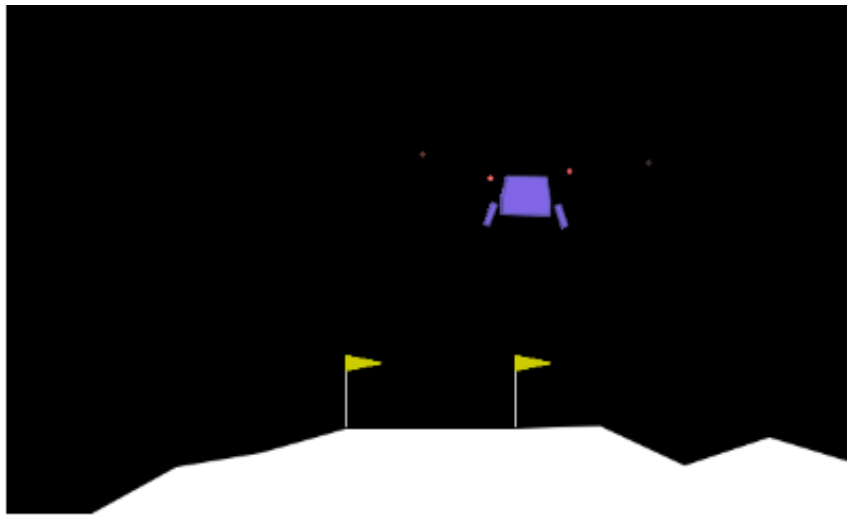
Installing Gymnasium

Execute in a a terminal (inside Anaconda, if you are using Anaconda):

```
pip install swig
pip install gymnasium[toy_text]
pip install gymnasium[box2d]
```

You can follow the link below for more detailed instructions:
https://github.com/Farama-Foundation/Gymnasium

The Lunar Lander Problem



A classic rocket trajectory optimization problem.

- Decision: Fire engine on or off.

Action Space:

- 0: do nothing
- 1: fire left orientation engine
- 2: fire main engine
- 3: fire right orientation engine

The Lunar Lander Problem (cont'd)

Observation Space:
An 8-dimensional vector:

- The coordinates of the lander in x and y
- Linear velocities in x and y
- Its angle
- its angular velocity
- 2 booleans that represent whether each leg is in contact with the ground or not.

The Lunar Lander Problem (cont'd)

Rewards:

After every step a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode.

For each step, the reward:

- is increased/decreased the closer/further the lander is to the landing pad.
- is increased/decreased the slower/faster the lander is moving.
- is decreased the more the lander is tilted (angle not horizontal).
- is increased by 10 points for each leg that is in contact with the ground.
- is decreased by 0.03 points each frame a side engine is firing.
- is decreased by 0.3 points each frame the main engine is firing.

The episode receive an additional reward of -100 or +100 points for crashing or landing safely respectively.

$\longrightarrow$ An episode is considered a solution if it scores at least 200 points.

The Lunar Lander Problem (cont'd)

Episode Termination:

- the lander crashes (the lander body gets in contact with the moon)
- the lander gets outside of the viewport (x coordinate is greater than 1);

Example of Robot Agent (Random Policy) for Lunar Landing

```
import gymnasium as gym

#env = gym.make("LunarLander-v3", render_mode="human")
env = gym.make("LunarLander-v3", render_mode="rgb_array")
observation, info = env.reset()

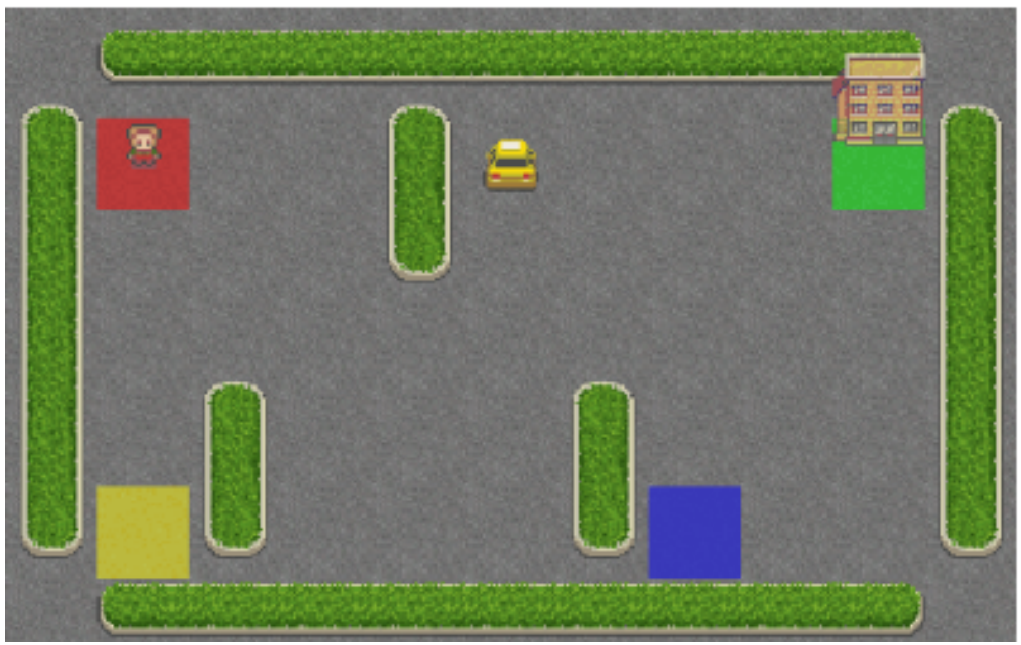
total_reward = 0
episode_over = False
while not episode_over:
    # agent random policy that uses the observation and info
    action = env.action_space.sample()
    observation, reward, terminated, truncated, info = env.step(action)

    print(f"Action: {action} - Observation: {observation}, reward: {reward}")
    total_reward += reward
    episode_over = terminated or truncated

print(total_reward)
env.close()
```

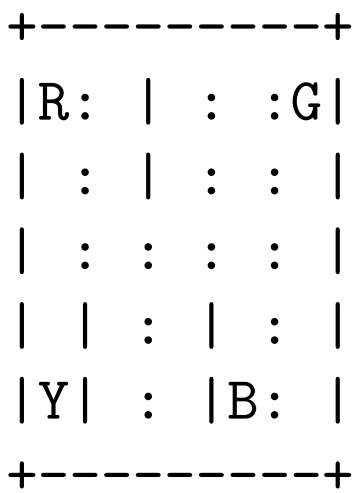
The Taxi Driver Problem

Trying to find the optimum path to a passenger location, pick up the passenger and then travel to a goal location and drop-off the passenger.



The Taxi Driver Problem (cont'd)

The discrete environment is a 5x5 maze.



- There are four designated pick-up and drop-off locations (Red, Green, Yellow and Blue) in the 5x5 grid world. The taxi starts off at a random square and the passenger at one of the designated locations.
- The goal for the robot is to drive the taxi to the passenger's location, pick up the passenger, move to the passenger's desired destination, and drop off the passenger. Once the passenger is dropped off, the episode ends.

The Taxi Driver Problem (cont'd)

Action Space:

- 0: Move south (down)
- 1: Move north (up)
- 2: Move east (right)
- 3: Move west (left)
- 4: Pickup passenger
- 5: Drop off passenger

The Taxi Driver Problem (cont'd)

Rewards at each time step:

- -1 per step unless other reward is triggered.
- $+20$ delivering passenger.
- -10 executing "pickup" and "drop-off" actions illegally.
- An action that results a noop (no operation), like moving into a wall, will incur the time step penalty.

Episode end:

- Termination: 1. The taxi drops off the passenger.
- Truncation: 2. The length of the episode is 200.

The Taxi Driver Problem (cont'd)

Observation is returned as an integer that encodes the corresponding state, calculated by:

$$((taxi_row * 5 + taxi_col) * 5 + passenger_location) * 4 + destination$$

Encoding:

```
def encode(taxi_row, taxi_col, pass_loc, dest_idx):
    # possible values for each argument: (5) 5, 5, 4
    i = taxi_row
    i *= 5
    i += taxi_col
    i *= 5
    i += pass_loc
    i *= 4
    i += dest_idx
    return i
```

The Taxi Driver Problem (cont'd)

Observation is returned as an integer that encodes the corresponding state, calculated by:

$$((taxi_row * 5 + taxi_col) * 5 + passenger_location) * 4 + destination$$

Decoding:

```
def decode(i):
    out = []

    out.append(i % 4)
    i = i // 4
    out.append(i % 5)
    i = i // 5
    out.append(i % 5)
    i = i // 5
    out.append(i)
    assert 0 <= i < 5

    return reversed(out)
```

Example of Robot Agent (Random Policy) for the Taxi Driver Problem

```python
import gymnasium as gym

env = gym.make("Taxi-v3", render_mode="rgb_array")
observation, info = env.reset()

total_reward = 0
episode_over = False
while not episode_over:
    # robot random policy that uses the observation and info
    action = env.action_space.sample()
    observation, reward, terminated, truncated, info = env.step(action)

    print(f"Action: {action} - Observation: {observation}, reward: {reward}")
    total_reward += reward
    episode_over = terminated or truncated

print(f'\n--> Total reward: {total_reward}')
env.close()
```