Reinforcement Learning

Assignment #01

SAPIENZA
UNIVERSITÀ DI ROMA

Info

- **Deadline:** Wednesday November 8th

- Students may discuss assignments, but the solutions must be typed and coded up **individually**

- Students must **indicate the names of colleagues they collaborated with**

Folder organization

- The assignment source code will be available on Classroom. You will find:
  - assignment1.pdf: with all the information
  - assignment1.zip that contains:
    - requirements.txt
    - - ilqr/
      - main.py (do not touch!)
      - student.py
    - policy_iteration/
      - main.py (do not touch!)
      - policy_iteration.py (do not touch!)
      - student.py

Theory submission

The theory solutions must be submitted in a pdf file named “XXXXXXX.pdf”, where XXXXXXX is your matricula.

We encourage you to type equations on an editor, rather than uploading scanned files.

Use the pdf file also to communicate the **students you collaborated with** and to insert a **small report of the code exercises**.

Code submission

The code solutions must be submitted in a zip file named “XXXXXXX.zip”, where XXXXXXX is your matricula.

The zip file must be organized exactly as the original assignment.zip file. Wrongly submitted assignments will be penalized.

Only edit the “students.py” files.

Theory

1. Derive the formula to get the minimum number of iterations that are needed if we want an ϵ error on the quality of the policy we get with Value Iteration algorithm.
2. Given the following environment settings
   - States: $\{s_i : \ i \in \{1...7\}\}$
   - Reward:
   $$r(s,a) = \begin{cases} 0.5 & \text{if } s = s_1 \\ 5 & \text{if } s = s_7 \\ 0 & \text{otherwise} \end{cases}$$
   - Dynamics: $p(s_6|s_6,a_1) = 0.3$, $p(s_7|s_6,a_1) = 0.7$
   - Policy: $\pi(s) = a_1 \ \ \forall s \in S$

   and the value function at the iteration $k = 1$:

   $$v_k = [0.5,\ 0,\ 0,\ 0,\ 0,\ 0,\ 5],$$

   with $\gamma = 0.9$.

   Compute $V_{k+1}(s_6)$ following the *Value Iteration* algorithm.

Code: Policy iteration

```python
def reward_function(s, env_size):
    r = ... # ?

    return r
```

1. Implement the reward function as described in Gymnasium documentation.

It should return 1 in the goal and 0 everywhere else.

Code: Policy iteration

```python
def check_feasibility(s_prime, s, env_size, obstacles):
    # TODO

    return s
```

2. Implement the transition function. It should return s_prime if it is a feasible state, s otherwise.

Code: Policy iteration

```python
def transition_probabilities(env, s, a, env_size, directions, obstacles):
    cells = []
    probs = []
    prob_next_state = np.zeros((env_size, env_size))

    # TODO
    s_prime = ... # ??
    prob_next_state[s_prime[0], s_prime[1]] = ... # ???

    s_prime = ... # ??
    prob_next_state[s_prime[0], s_prime[1]] = ... # ???

    s_prime = ... # ??
    prob_next_state[s_prime[0], s_prime[1]] = ... # ???

    return prob_next_state
```

3. Implement the function returning the transition probabilities.

For each possible next state s_prime, add the probability of ending up in that state in the prob_next_state matrix.

Code: iLQR

```python
def pendulum_dyn(x,u):
    th = x[0]
    thdot = x[1]

    g = 10.
    m = 1.
    l = 1.
    dt = 0.05

    u = np.clip(u, -2, 2)[0]

    # TODO
    newthdot = ...
    newth = ...

    newthdot = np.clip(newthdot, -8, 8)

    x = np.array([newth, newthdot])
    return x
```

1. Implement the pendulum dynamics as described in Gymnasium documentation.

$$\dot{\theta}_{t+1} = \dot{\theta}_t + \left(\frac{3g}{2l} \sin\theta + \frac{3.0}{ml^2} u \right) dt$$

$$\theta_{t+1} = \theta + \dot{\theta}_{t+1} \, dt$$

Code: iLQR

---

2. Write the equations to compute the K and P matrices of the LQR-LTV algorithm. Note that in this case, there are some additional terms coming from the 2° order Taylor expansion of the cost, that can be computed using the following formulas:

$$k_t = -(R_t + B_t^T P_{t+1} B_t)^{-1}(r_t + B_t^T p_{t+1})$$

$$p_t = q_t + K^T(R_t k_t + r_t) + (A_t + B_t K)^T p_t + (A_t + B_t K)^T P_{t+1} B_t k_t$$

```python
def backward(self, x_seq, u_seq):

    pt1 = self.getq(x_seq[-1],u_seq[-1])
    Pt1 = self.getQ(x_seq[-1],u_seq[-1])

    k_seq = []
    K_seq = []

    for t in range(self.horizon-1,-1,-1):

        xt = x_seq[t]
        ut = u_seq[t]

        At = self.getA(xt,ut)
        Bt = self.getB(xt,ut)

        qt = self.getq(xt,ut)
        rt = self.getr(xt,ut)

        Qt = self.getQ(xt,ut)
        Rt = self.getR(xt,ut)

        # TODO
        kt = ...
        Kt = ...
        # TODO
        pt = ...
        Pt = ...

        pt1 = pt
        Pt1 = Pt

        k_seq.append(kt)
        K_seq.append(Kt)

    k_seq.reverse()
    K_seq.reverse()

    return k_seq,K_seq
```

Code: iLQR

```python
def forward(self, x_seq, u_seq, k_seq, K_seq):

    x_seq_hat = np.array(x_seq)
    u_seq_hat = np.array(u_seq)

    for t in range(len(u_seq)):
        # TODO
        control = ...

        # clip controls to the actual range from gymnasium
        u_seq_hat[t] = np.clip(u_seq[t] + control,-2,2)
        x_seq_hat[t+1] = self.f(x_seq_hat[t], u_seq_hat[t])

    return x_seq_hat, u_seq_hat
```

3. Given the computed k_t and K_t, compute the control as:

$$control = k_t + K_t(x_t^i - x_t^{i-1})$$