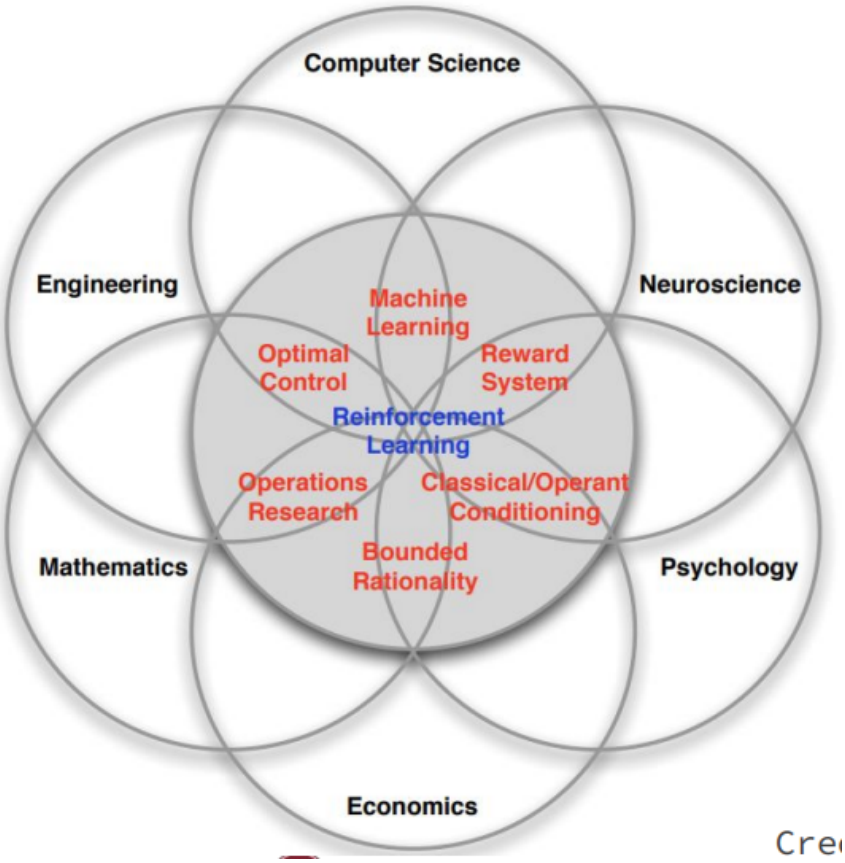


Reinforcement Learning

*H*omework 1

Yusupha Juwara
juwara.1936515@studenti.uniroma1.it

November 8, 2023

Contents

Chapter 1

Theory Part

Question 1.1: Derive the formula to get the minimum number of iterations of Value Iteration that are needed if we want an error on the quality of the policy that is at most ϵ.

Answer 1.1: We know that $\pi^* = \text{argmax}_a Q^*(s,a)$, and since $Q_i(s,a) \cong Q^*(s,a)$ we could choose:

$$\pi_i(s) = \text{argmax}_a Q^*(s,a)$$

What is the quality of such a policy? For all states

$$V^{\pi^i}(s) \geq V^*(s) - \frac{2\gamma^i}{1-\gamma}||Q_0 - Q^*||$$

Proof:

$$V^{\pi^i}(s) - V^*(s) = Q^{\pi^i}(s,\pi^i(s)) - Q^*(s,\pi^*(s))$$

$$= Q^{\pi^i}(s,\pi^i(s)) \overbrace{-Q^*(s,\pi^i(s)) + Q^*(s,\pi^i(s))}^{\text{add \& subtract}} - Q^*(s,\pi^*(s))$$

$$= \overbrace{Q^{\pi^i}(s,\pi^i(s)) - Q^*(s,\pi^i(s))}^{\text{expand \& get rid of r}} + Q^*(s,\pi^i(s)) - Q^*(s,\pi^*(s))$$

$$= \gamma \mathbb{E}_{s' \sim P(s,\pi^i(s))}\left(V^{\pi^i}(s') - V^*(s')\right) + Q^*(s,\pi^i(s)) - Q^*(s,\pi^*(s))$$

$$\geq \gamma \mathbb{E}_{s' \sim P(s,\pi^i(s))}\left(V^{\pi^i}(s') - V^*(s')\right) \overbrace{+Q^*(s,\pi^i(s)) - Q^i(s,\pi^i(s))}^{\text{by definition of } Q^*}$$

$$\overbrace{+Q^i(s,\pi^*(s)) - Q^*(s,\pi^*(s))}^{\text{by definition of } Q^*}$$

$$\geq \gamma \mathbb{E}_{s' \sim P(s,\pi^i(s))}\left(\overbrace{V^{\pi^i}(s') - V^*(s')}^{\text{repeat \& get } \frac{1}{1-\gamma}}\right) - 2 \overbrace{\gamma^i||Q_0 - Q^*||_\infty}^{||Q_i - Q^*|| \leq \gamma^i ||Q_0 - Q^*||}$$

If we want an ϵ error on the quality of the policy to determine the number of iterations i, we can just solve for it in this equation:

$$\frac{2\gamma^i}{1-\gamma}||Q_0 - Q^*|| \leq \epsilon$$

Note that both Q_0 and Q^* are in the range $[0, \frac{1}{1-\gamma}]$, and, so, by the infinity norm, the max value of $||Q_0 - Q^*||$ is $\frac{1}{1-\gamma}$.

$$\frac{2\gamma^i}{1-\gamma}||Q_0 - Q^*|| = \frac{2\overbrace{(1-(1-\gamma^i))}^{\leq 1}}{1-\gamma}||Q^*|| \leq \epsilon$$

$$\implies \frac{2\overbrace{e^{-i(1-\gamma)}}^{1+x \leq e^x \ \forall x \in \mathbb{R}}}{(1-\gamma)^2} \leq \epsilon$$

$$\implies e^{-i(1-\gamma)} \leq \frac{\epsilon(1-\gamma)^2}{2}$$

$$\implies -i(1-\gamma) \leq \overbrace{-log(\frac{2}{\epsilon(1-\gamma)^2}}^{log(\frac{1}{x}) = -log(x)}$$

$$\implies i \geq \frac{log\left(\frac{2}{\epsilon(1-\gamma)^2}\right)}{1-\gamma}$$

Note that i does not depend on the states and actions.

Question 1.2: Given the following environment settings:

- **States:** $S = \{s_i : i \in \{1, \cdots, 7\}\}$
- **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:** $p(s_6 \mid s_6, a_1) = 0.3, \ p(s_7 \mid s_6, a_1) = 0.7$
- **Policy:** $\pi(s) = a_1, \ \forall s \in S$
- **Value function** at the iteration $k = 1$:
$$v_k = [0.5, 0, 0, 0, 0, 0, 5],$$
- $\gamma = 0.9$

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

Answer 1.2:

$$V_{k+1}(s) = \max_a \left[r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s, a) V_k(s')\right]$$

$$V_{k+1}(s_6) = V_2(s_6) = r(s) + \gamma \max_a \left[\sum_{s' \in S} \overbrace{p(s' \mid s, a)}^{0 \ \forall s' \notin \{s_6, s7\}} V_k(s')\right]$$

$$V_2(s_6) = 0 + 0.9\left[\overbrace{p(s_6 \mid s_6, a_1)}^{0.3}\overbrace{V_1(s_6)}^{0} + \overbrace{p(s_7 \mid s_6, a_1)}^{0.7}\overbrace{V_1(s_7)}^{5}\right]$$

$$V_2(s_6) = 0 + 0.9\left[0.3 \times 0 + 0.7 \times 5\right]$$

$$V_2(s_6) = 3.15$$

Notice that $r(s, a) = r(s)$ because it is independent of the next state and action; so, it is not multiplied by the transition probability $p(s' \mid s, a)$.

Chapter 2

Code Report

2.1 Introduction

This report presents the implementation of two reinforcement learning algorithms, **Policy Iteration** for the Frozen Lake environment and **iLQR** (**Iterative Linear Quadratic Regulator**) for the Pendulum environment. The goal is to solve these problems using Gymnasium for environment modeling.

2.2 Policy Iteration for Frozen Lake

2.2.1 Frozen Lake Overview

The Frozen Lake environment involves crossing a frozen lake from the start to the goal without falling into holes. The player may move in unintended directions due to the slippery nature of the frozen lake. Furthermore, due to the **Homework Assignment specifications**, the agent can move in the right direction with only a probability of $\frac{1}{3}$, while the other two-thirds is shared equally between the perpendicular directions from the agent's current position.

2.2.2 Reward Function

The reward function is defined as follows:

$$\text{reward}(s) = \begin{cases} 1 & \text{if } s \text{ is the goal state} \\ 0 & \text{otherwise} \end{cases}$$

This reward function assigns a reward of 1 to the goal state and 0 to all other states.

```
# Get the correct reward for each state; 1 for the goal state, 0 for all the rest
def reward_function(s, env_size):
    r = 0.0
    END_STATE = np.array([env_size-1, env_size-1], dtype=np.uint8)
    if (s == END_STATE).all():
        r = 1
    return r

# Call the reward_function to obtain and assign the correct rewards to their respective states
def reward_probabilities(env_size):
    rewards = np.zeros((env_size*env_size))
    i = 0
    for r in range(env_size):
        for c in range(env_size):
            state = np.array([r,c], dtype=np.uint8)
            rewards[i] = reward_function(state, env_size)
            i+=1
    return rewards
```

This means that in code, we initialize the "rewards" as all-zeros and then modify only the index position that corresponds to the goal state.

2.2.3 Check Feasibility of Transitions

The check_feasibility function ensures that the next state is feasible; check feasibility of the new state, if it is a possible state return s_prime, otherwise return s. It does this by checking that all the next possible states are within the bounds of the grid. If that is not true, then it returns the agent's current state.

$$\text{transition}(s) = \begin{cases} s_{\text{prime}} & \text{if } s_{\text{prime}} \text{ is a feasible state} \\ s & \text{otherwise} \end{cases}$$

```
def check_feasibility(s_prime, s, env_size, obstacles):

    # Check that the agent is within the bounds of the grid
    if s_prime[0] < env_size and s_prime[1] < env_size and (s_prime >= 0).all():
        return s_prime

    return s
```

2.2.4 Transition Probabilities

This is the function that utilizes the check_feasibility function above. For all states, it computes the transition probabilities of going to a next state given the current state and action. Thus, it does two things:

- Check the feasibility of going to a next state.
- assign a probability of $\frac{1}{3}$ for the state where the "right" action goes to. Then, the remaining $\frac{2}{3}$ is equally divided and assigned to the **perpendicular** directions.

```
def transition_probabilities(...)
    ...
    # Get the next state with action 'a'
    s_prime = s + directions[a, :]

    # Check its feasibility as defined above.
    s_prime = check_feasibility(s_prime, s, env_size, obstacles)

    # If s_prime is differnt from the current state, then assign prob of 1/3
    if (s != s_prime).any():
        prob_next_state[s_prime[0], s_prime[1]] = prob
        prob_sum += prob
```

```
    # Get the next state with the next action 'a+1' in the action space
    s_prime = s + directions[a, :]
    # Then do as from lines 6) to 12)

    # Get the next state with the previous action 'a -1' in the action space
    s_prime = s + directions[a, :]
    # Then do as from lines 6) to 12)

    # Assign the remaining probs to the current state
    prob_next_state[s[0], s[1]] = 1-prob_sum
    ...
```

Notice that the 1 - prob_sum makes sure that the current state gets its due share and the overall probabilities sum to 1.

2.2.5 Policy Iteration Algorithm

Equation for Policy Evaluation:

$$V^{\pi}(s) = \sum_{s',r} P(s', r|s, \pi(s)) \cdot [r + \gamma \cdot V^{\pi}(s')]$$

Equation for Policy Improvement:

$$\pi'(s) = \text{argmax}_a \sum_{s',r} P(s', r|s, a) \cdot [r + \gamma \cdot V^{\pi}(s')]$$

Code Snippet for Policy Iteration:

```
# Policy Iteration Algorithm
def policy_iteration(env):
    # Initialize policy and value function
    ...
    while True or max iters:
        # Do Policy Evaluation
        V = evaluate_policy(policy, ...)

        # Do Policy Improvement
        new_policy = improve_policy(V, ...)

        # Check if the new policy is the same as the old, then break
        if new_policy == policy:
            break

        policy = new_policy

    return policy, V
```

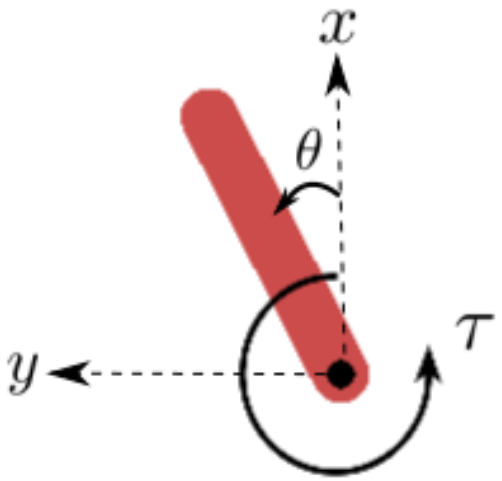
2.3 iLQR for Pendulum

2.3.1 Pendulum Overview

From the documentation, the inverted pendulum swingup problem is based on the classic problem in control theory. The system consists of a pendulum attached at one end to a fixed point, and the other end being free. The pendulum starts in a random position and the goal is to apply torque on the free end to swing it into an upright position, with its center of gravity right above the fixed point.

The diagram below specifies the coordinate system used for the implementation of the pendulum's dynamic equations.

- $x - y$: cartesian coordinates of the pendulum's end in meters.
- θ : angle in radians.
- τ: torque in N m. Defined as positive counter-clockwise.



- The **action space** represents the torque applied to free end of the pendulum between [-2,2].
- The **environment space** represents the $x - y$ coordinates of the pendulum's free end and its angular velocity.
  - $x = \cos(\theta) \in [-1, 1]$
  - $y = \sin(\theta) \in [-1, 1]$
  - Angular Velocity $\in [-8, 8]$

2.3.2 Pendulum Dynamics

The dynamics model the behavior of a pendulum, taking into account its angle and angular velocity. The pendulum dynamics are represented by the equations:

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

$$\theta_{t+1} = \theta_t + \dot{\theta}_{t+1} \cdot \Delta t$$

$$\dot{\theta}_{t+1} = \text{clip}(\dot{\theta_{t+1}}, -8, 8)$$

Code Snippet for the Pendulum Dynamics:

```
def pendulum_dyn(x,u):
    ...
    num_1 = np.sin(th)*3*g
    den_1 = 2*l
    term_1 = num_1 / den_1
    num_2 = 3.0*u
    den_2 = m*(l**2)
    term_2 = num_2 / den_2

    newthdot = thdot + (term_1 + term_2) * dt
    newth = th + newthdot*dt

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

2.3.3 Equations for K, P Matrices and Cost Linearization terms k and p

In the **backward function**, the equations for computing the K (control gains) and P (cost-to-go) matrices for the LQR-LTV algorithm are as below. These equations calculate the control gains (K) and cost-to-go (P) matrices based on the current state and estimated system dynamics. In this problem, there are also additional terms k_t and p_t coming from the cost's linearization, performed using the 2^{nd} order Taylor expansion.

$$
\begin{aligned}
k_t &= -(R_t + B_t^T P_{t+1} B_t)^{-1} \cdot (r_t + B_t^T P_{t+1}) \\
K_t &= -(R_t + B_t^T P_{t+1} B_t)^{-1} \cdot B_t^T P_{t+1} \cdot A_t \\
p_t &= q_t + K_t^T (R_t \cdot k_t + r_t) + (A_t + B_t K_t)^T p_{t+1} \\
&\quad + (A_t + B_t K_t)^T P_{t+1} B_t k_t \\
P_t &= Q_t + K_t^T R_t K_t + (A_t + B_t K_t)^T P_{t+1} (A_t + B_t K_t)
\end{aligned}
$$

Code Snippet for the K, P Matrices and Cost Linearization terms k and p:

```
def backward(self, x_seq, u_seq):
    ...
    kt = -linalg.inv(Rt+Bt.T@Pt1@Bt)@(rt+Bt.T@pt1)  # shape: (1,)
    Kt = -linalg.inv(Rt+Bt.T@Pt1@Bt)@Bt.T@Pt1@At    # shape: (1, 2)

    pt = qt + Kt.T@(Rt@kt+rt)+(At+Bt@Kt).T@pt1+(At+Bt@Kt).T@Pt1@Bt@kt
    Pt = Qt + Kt.T@Rt@Kt + (At+Bt@Kt).T@Pt1@(At+Bt@Kt)
    ...
```

2.3.4 Control

Once the K and P matrices have been computed, the control input for the pendulum system can be determined. The control input is calculated in the **forward function** using **feedback control law**, incorporating the computed control gains (K) and state estimation.

Equation for Control Calculation:

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

Code Snippet for the K, P Matrices and Cost Linearization terms k and p:

```
def forward(self, x_seq, u_seq, k_seq, K_seq):
    ...
    # Compute control using the feedback control law
    control = k_seq[t] + K_seq[t]@(x_seq_hat[t] - x_seq[t])
    ...
```

2.4 Conclusion

The implementations of Policy Iteration for the Frozen Lake environment and iLQR for the Pendulum showcase effective solutions to reinforcement learning challenges. This report has provided valuable insights into addressing control problems through the application of the feedback control law within Gymnasium's Pendulum environment. Additionally, it has delved into Policy Iteration, using Gymnasium's Frozen Lake environment, where the problem was creatively reformulated to model diverse scenarios. In these scenarios, the agent faces the task of executing the current action while also dealing with the added complexity of potential unintended movement into any of the perpendicular directions, each with equal probability. This augmentation in difficulty enriches the agent's learning experience.