Assignment 1

Reinforcement Learning Programming - CSCN 8020

By: Jarius Bedward

Link to Repo:

https://github.com/JariusBedward-8841640/CSCN8020_A1.git

February 9, 2026

Problem 1 [10]

Pick-and-Place Robot: Consider using reinforcement learning to control the motion of a robot arm in a repetitive pick-and-place task. If we want to learn movements that are fast and smooth, the learning agent will have to control the motors directly and obtain feedback about the current positions and velocities of the mechanical linkages.
Design the reinforcement learning problem as an MDP, define states, actions, rewards *with reasoning*.

To control a robotic arm performing a repetitive pick and place task using RL, we first will model the problem as a Markov Decision Process (MDP). The objective is to learn the motor movements that are both fast and smooth while accurately completing the task. The MDP is defined by the tuple:

(S, A, P, R, γ)

Where S represents states, A actions, P transition probabilities, R Rewards and γ the discount factor

State (S)

The state must fully describe the robots current configuration so that the Markov property is satisfied

The state includes:

- Joint angles (positions): $\theta_1, \theta_2, .., \theta_n$
- Joint angular velocities: $\dot{\theta}_1, \dot{\theta}_2, .., \dot{\theta}_n$
- Grasp status (object held or not)
- Target position

The state can be represented as: $s \doteq (\theta_{1\ldots}, \theta_n, \dot{\theta}_1, .., \dot{\theta}_n, g, xtarget)$

We have to include velocities is necessary because smoothness depends on changes in motion, and without velocities, the system would not satisfy the Markov property.

Action (A)

The agent directly controls the robot’s motors0-

The action consists of continuous motor torques applied to each joint $\alpha=(\tau_{1,}\tau_2,...,\tau_n)$

Where τ_i represents the torque applied at join .

A continuous action space allows the agent to learn precise and smooth motor control

Reward Function (R)

The reward function must encourage fast task completion, smooth motion and acuurate placement

The reward at each timestep can be composed of

Success Reward

A large positive reward when the object is successfully placed:

$R_{success}=+100$

Time penalty

A small negative reward at each timestep to encourage faster completion

$R_{time}=-c$

Smoothness

Penalizing large torques to promote smooth movements

$R_{energy=-}\alpha\|\tau\|^2$

Distance penalty

Penalizing distance between end effector and target

$R_{distance=-}\beta\| x_{effector}-x_{target}\|$

The reward overall can be rewritten as

$R = R_{success} - c - \alpha\|\tau\|^2 - \beta\| x_{effector} - x_{target}\|$

Transition Function (P)

The transition function describes how the robot moves from one state to another based on the physics $s_{t+1}=f(s_t,a_t)$

The next state can depend on things like applied torques or gravity or friction

Discount Factor(γ)

Because this is an episodic task where long term performance matters a high discount factor should be appropriate

$\gamma \approx 0.95 – 0.99$

This encourages the agent to optimize the entire trajectory rather than only immediate rewards

Conclusion

By modeling the problem as an MDP, reinforcement learning can learn efficient and stable motor control policies for repetitive manipulation tasks

Problem 2 [20]

Problem Statement

2x2 Gridworld: Consider a 2x2 gridworld with the following characteristics:

- State Space (S): s_1,s_2,s_3,s_4.
- Action Space (A): up, down, left, right.
- Initial Policy (π): For all states, $\pi(up|s) = 1$.
- Transition Probabilities $P(s'|s,a)$:

  - If the action is valid (does not run into a wall), the transition is deterministic.
  - Otherwise, $s' = s$.

- Rewards $R(s)$:
  - $R(s_1) = 5$ for all actions a.

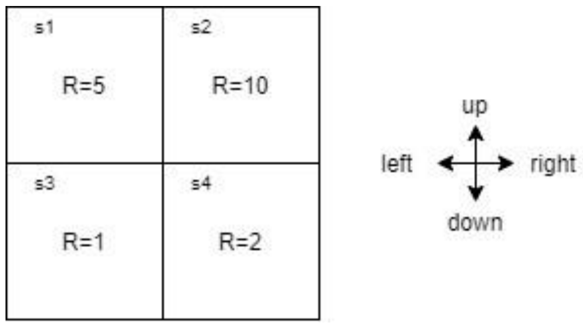


Figure 1: 2x2 Gridworld

  - $R(s_2) = 10$ for all actions a.
  - $R(s_3) = 1$ for all actions a.
  - $R(s_4) = 2$ for all actions a.

Tasks

Perform two iterations of Value Iteration for this gridworld environment. Show the step-by-step process (**without code**) including policy evaluation and policy improvement. Provide the following for each iteration:

- Iteration 1:
  1. Show the initial value function (V) for each state.

$$\boldsymbol{V_0(s) = 0 \ \forall s}$$

This means at this iteration all the states values are 0

State	$\boldsymbol{V_0(s)}$
s1	0
s2	0
s3	0
s4	0

2. Perform value function updates.

   For $k = 0$:

$$\boldsymbol{V_1(s)} = \backslash\boldsymbol{max}\ \_a[R(s) + V_0(s')]$$

   Since $\boldsymbol{V_0(s') = 0}$

$$\boldsymbol{V_1(s)} = \backslash\boldsymbol{max}\ \_a[R(s)]$$

   Because rewards depend only on the states and are identical for all actions

   Ex. For state s_1

$$\boldsymbol{R(s_1) = 5}$$

$$\boldsymbol{V_1(s_1)} = \backslash\boldsymbol{max}\ \_a[5] = \boldsymbol{5}$$

3. Show the updated value function.

State	$\boldsymbol{V_1(s)}$
s1	5
s2	10
s3	1
s4	2

- Iteration 2: Show the value function (V) after the second iteration.

Now we compute

$$\boldsymbol{V_2(s)} = \backslash\boldsymbol{max}\ \_a[R(s) + V_1(s')]$$

State s_1
Possible next states

Right > $s_2(10)$
Down > s_3 (1)
Up/Left > $s_1(5)$

Best next value = 10

$$\boldsymbol{V_2(s1) = 5 + 10 = 15}$$

State s_2
Possible next states

Down > $s_4(2)$
Left > s_1 (5)
Up/Right> $s_2(10)$

Best next value = 10

$$\boldsymbol{V_2(s1) = 10 + 10 = 20}$$

State s_3
Possible next states

Up > $s_1(5)$
Right > s_4 (2)
Down/Left > $s_3(1)$

Best next value = 10

$$\boldsymbol{V_2(s3) = 1 + 5 = 6}$$

State s_5
Possible next states

Up > $s_2(10)$
Left > s_3 (1)
Down/Right > $s_1(2)$

Best next value = 10

$$\boldsymbol{V_2(s1) = 2 + 10 = 12}$$

State	$\boldsymbol{V_2(s)}$
s1	15
s2	20
s3	6
s4	12

Problem 3 [35]

Problem Statement

5x5 Gridworld: In Lecture 3's programming exercise (here), we explored an MDP based on a 5x5 gridworld and implemented Value Iteration to estimate the optimal state-value function (V_*) and optimal policy (π_*).
The environment can be described as follows:

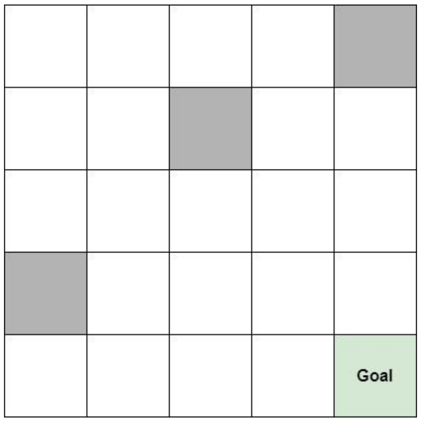


Figure 2: 5x5 Gridworld

- States: states are identified by their row and column, the same as a regular matrix. Ex: the state in row 0 and column 3 is $s_{0,3}$ (Figure: 2)
  - Terminal/Goal state: The episode ends if the agent reached this state. $s_{Goal} = s_{4,4}$
  - Grey states: $\{s_{2,2}, s_{3,0}, s_{0,4}\}$, these are valid but non-favourable states, as will be seen in the reward function.
- Actions: a_1 = right, a_2 = down, a_3 = down, a_4 = up for all states.
- Transitions: If an action is valid, the transition is deterministic, otherwise $s' = s$
- Rewards $R(s)$:

$$\begin{aligned} R(s) &= +10 \qquad s = s_{4,4} \\ &= -5 \qquad s \in S_{grey} = s_{2,2}, s_{3,0}, s_{0,4} \\ &= -1 \qquad s \in S /= s_{4,4}, S_{grey} \end{aligned}$$

Tasks

Task1: Update MDP Code

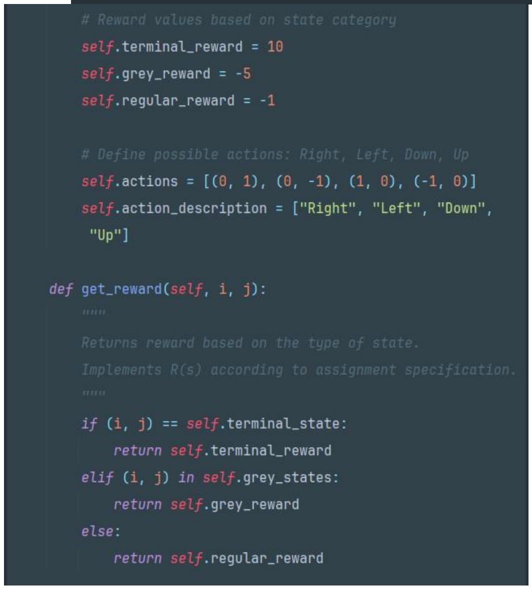
1. Update the reward function to be a list of reward based on whether the state is terminal, grey, or a regular state.

```
2.  # Task 1: Updating the Reward Function
- In an MDP, the reward function is formally defined as a mapping

- Instead of pre-filling a reward matrix we update the code by implementing
```

```
the rule directly into the environment to reflect the mathematical
definition

- The reward is implemented dynamically based on state category instead of
storing a fixed reward grid. This is done to make sure that there is a clean
separation between environment structure and reward logic as well as easy
modification if reward values change
```



```python
        # Reward values based on state category
        self.terminal_reward = 10
        self.grey_reward = -5
        self.regular_reward = -1

        # Define possible actions: Right, Left, Down, Up
        self.actions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        self.action_description = ["Right", "Left", "Down",
         "Up"]

    def get_reward(self, i, j):
        """
        Returns reward based on the type of state.
        Implements R(s) according to assignment specification.
        """
        if (i, j) == self.terminal_state:
            return self.terminal_reward
        elif (i, j) in self.grey_states:
            return self.grey_reward
        else:
            return self.regular_reward
```

2. Run the existing code developed in class and obtain the optimal state-values and optimal policy. Provide a figures of the gridworld with the obtained V_* and π_* (You can manually create a table).

Optimal Value Function (V*)

	0	1	2	3	4
0	-0.43	0.63	1.81	3.12	4.58
1	0.63	1.81	3.12	4.58	6.2
2	1.81	3.12	4.58	6.2	8.0
3	3.12	4.58	6.2	8.0	10.0
4	4.58	6.2	8.0	10.0	0.0

Optimal Policy (π*)”

	0	1	2	3	4
0	Right	Right	Right	Down	Down
1	Right	Right	Right	Right	Down
2	Right	Down	Right	Right	Down
3	Right	Right	Right	Right	Down
4	Right	Right	Right	Right	Goal

(Interpretation in code markdown)

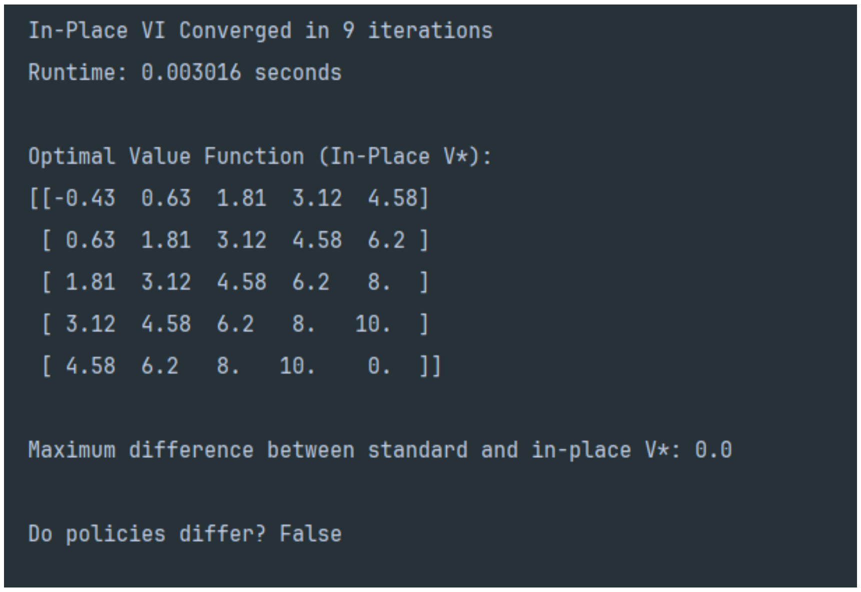
Task 2: Value Iteration Variations

Implement the following variation of value iteration. Confirm that it reaches the same optimal statevalue function and policy.

1. **In-Place Value Iteration**: Use a single array to store the state values. This means that you update the value of a state and immediately use that updated value in the subsequent updates.

   (More talking points in code markdown)

Output:


```
In-Place VI Converged in 9 iterations
Runtime: 0.003016 seconds

Optimal Value Function (In-Place V*):
[[-0.43  0.63  1.81  3.12  4.58]
 [ 0.63  1.81  3.12  4.58  6.2 ]
 [ 1.81  3.12  4.58  6.2   8.  ]
 [ 3.12  4.58  6.2   8.   10.  ]
 [ 4.58  6.2   8.   10.    0.  ]]

Maximum difference between standard and in-place V*: 0.0

Do policies differ? False
```

Deliverables

- Full code with comments to explain key steps and calculations.

- Provide the estimated value function for each state.

- **Important:** Compare the performance of these variations in terms of optimization time, number of episodes, and provide comments on their computational complexity.

Problem 4 [35]

Problem Statement

Off-policy Monte Carlo with Importance Sampling: We will use the same environment, states, actions, and rewards in Problem 3.

Task

Implement the off-policy Monte Carlo with Importance sampling algorithm to estimate the value function for the given gridworld. Use a fixed behavior policy $b(a|s)$ (e.g., a random policy) to generate episodes and a greedy target policy.

In this, episode are generated using a fixed behavior policy $b(a|s)$**.** which in this case is a random policy. The goal is to learn and improve a different policy the target policy, which is greedy with respect to our value estimates (Greedy refers to choosing the action that gives the highest immediate estimated return according to the current value function).

Because the data is generated under one policy but evaluated by another, the distribution of actions doesn't match but to correct this importance sampling is applied, which reweights the returns so they reflect what would have happened if the target policy had been followed

For each state visted in an episode we complete the return:

This is the total discounted reward from time step t until episode ends

$$G_t = \sum_{k=t}^{T-1} \gamma^{k-t} R_{k+1}$$

Importance sampling ratio:

Since actions were chosen using random behavior policy but evaluated under the greedy target policy this is corrected using

$$W_t = \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}$$

Because the behavior policy is uniform random:

$$b(a \mid s) = \frac{1}{4}$$

If the greedy target action matches the sampled action, the ratio contributes:

$$\frac{1}{1/4} = 4$$

If the action does not match the greedy target, then:

$$\pi(a \mid s) = 0$$

and the weight becomes zero, so the trajectory stops updating

A random Behavior Policy is used to guarantee the exploration of the state space; without exploration some states may never be visited and the value can't be estimated

If sampled action does not match the greedy target action, then

$$\pi(a \mid s) = 0$$

Which makes the importance ratio zero and continuing would contribute nothing except for increasing variance

Suggested steps

1. Generate multiple episodes using the behavior policy $b(a|s)$.
2. For each episode, calculate the returns (sum of discounted rewards) for each state.
3. Use importance sampling to estimate the value function and update the target policy $\pi(a|s)$.
4. You can assume a specific discount factor (e.g., $\gamma = 0.9$) for this problem.
5. Use the same main algorithm implemented in lecture 4 in class.

Deliverables

- Full code with comments to explain key steps and calculations.
- Provide the estimated value function for each state.

- **Important** Compare the estimated value function obtained from Monte Carlo with the one obtained from Value Iteration in terms of optimization time, number of episodes, computational complexity, and any other aspects you notice.