

Reinforcement Learning - Assignment 1

Marco Turchetti
Student ID: 1996734

Theory

Exercise 1 - Demonstrate the convergence of the Policy Iteration

The proof assumes a finite Markov Decision Process (MDP), which means there is a finite set of states and actions. The Policy Iteration algorithm proceeds in iterations, generating a sequence of monotonically improving policies and value functions. Because the number of possible deterministic policies in a finite MDP is also finite, this process is guaranteed to converge to the optimal policy, π^* , in a finite number of steps.

To formally demonstrate the convergence of the values, it must be shown that the distance between the policy's value function V^{π_i} and the optimal value function V^* converges to zero as the iterations i increase. Specifically, the following inequality will be proven:

$$\|V^{\pi_i} - V^*\|_\infty \leq \gamma^i \|V^{\pi_0} - V^*\|_\infty$$

where i is the iteration number and γ is the discount factor.

The Policy Iteration Algorithm

The policy iteration algorithm is an iterative procedure composed of two repeating steps: a **Policy Evaluation** step and a **Policy Improvement** step. This process generates a sequence of monotonically improving policies and value functions, which converges to the optimal policy, π^* , and the optimal value function, V^* .

The Proof Strategy

One iteration t of the algorithm is as follows:

1. **Policy Evaluation:** Given a policy π_t , its exact value function V^{π_t} is computed by solving the Bellman expectation equation.
2. **Policy Improvement:** From V^{π_t} , the Q-values are computed. The next policy π_{t+1} is found by acting greedily at every state:

$$\pi_{t+1}(s) = \arg \max_a Q^{\pi_t}(s, a)$$

The new policy π_{t+1} is derived from a greedy improvement. Since this greedy action selection is applied for all future states, the value of the new policy $V^{\pi_{t+1}}$ is an improvement over the value of applying only a single greedy action.

This "single greedy step" value is equivalent to the **Bellman optimality operator**, T , applied to the previous value function, V^{π_t} .

Therefore, the proof requires establishing the following inequality:

$$V^{\pi_{t+1}}(s) \geq (TV^{\pi_t})(s)$$

This inequality connects the algorithmic steps to the convergence proof. The components are defined below.

1. The Right-Hand Side: $(TV^{\pi_t})(s)$ The right-hand side is the **Bellman optimality operator** (T) applied to the value function of the *old* policy, V^{π_t} . By definition:

$$(TV^{\pi_t})(s) = \max_a \left[r(s, a) + \gamma \sum_{s'} p(s'|s, a) V^{\pi_t}(s') \right]$$

This value is the expected return of taking the **single best action** from state s (the \max_a) and then **reverting to the old policy** π_t for all subsequent steps.

2. The Left-Hand Side: $V^{\pi_{t+1}}(s)$ The left-hand side is the **true value function** of the *new* policy, π_{t+1} . This value is computed in the next "Policy Evaluation" step. By definition:

$$V^{\pi_{t+1}}(s) = r(s, \pi_{t+1}(s)) + \gamma \sum_{s'} p(s'|s, \pi_{t+1}(s)) V^{\pi_{t+1}}(s')$$

This is the expected return from following the **new policy** π_{t+1} from state s for all subsequent steps.

3. Justification of the Inequality (Policy Improvement Theorem) The Policy Improvement Theorem guarantees this inequality. The left-hand side, $V^{\pi_{t+1}}$, represents a policy that is "always greedy" (greedy at all steps). The right-hand side, (TV^{π_t}) , represents a policy that is "greedy for one step" (greedy now, then reverting to π_t). A consistently greedy policy must yield a value greater than or equal to a one-step greedy policy.

Completing the Proof of Convergence

This inequality is now used to prove convergence.

Step 1: One-Step Contraction It will be shown that the distance to the optimal value function V^* shrinks at each iteration.

- We begin with the distance from the optimum (using the max-norm, $\|\cdot\|_\infty$). Since $V^* \geq V^{\pi_{t+1}}$, the absolute value is the difference:

$$\|V^* - V^{\pi_{t+1}}\|_\infty = \max_s (V^*(s) - V^{\pi_{t+1}}(s))$$

- Applying the key inequality, $V^{\pi_{t+1}}(s) \geq (TV^{\pi_t})(s)$:

$$\dots \leq \max_s (V^*(s) - (TV^{\pi_t})(s))$$

- Using the fixed-point property of V^* , which is $V^* = TV^*$:

$$\dots = \max_s ((TV^*)(s) - (TV^{\pi_t})(s)) = \|TV^* - TV^{\pi_t}\|_\infty$$

- Using the property that T is a **contraction mapping** with factor γ :

$$\|TV^* - TV^{\pi_t}\|_\infty \leq \gamma \|V^* - V^{\pi_t}\|_\infty$$

This proves the one-step contraction:

$$\|V^* - V^{\pi_{t+1}}\|_\infty \leq \gamma \|V^* - V^{\pi_t}\|_\infty$$

Step 2: Unrolling the Recursion This one-step result is applied recursively from iteration i back to iteration 0.

$$\begin{aligned}
||V^{\pi_i} - V^*||_\infty &\leq \gamma ||V^{\pi_{i-1}} - V^*||_\infty \\
&\leq \gamma (\gamma ||V^{\pi_{i-2}} - V^*||_\infty) = \gamma^2 ||V^{\pi_{i-2}} - V^*||_\infty \\
&\leq \gamma^3 ||V^{\pi_{i-3}} - V^*||_\infty \\
&\quad \dots \\
&\leq \gamma^i ||V^{\pi_0} - V^*||_\infty
\end{aligned}$$

This final inequality, $||V^{\pi_i} - V^*||_\infty \leq \gamma^i ||V^{\pi_0} - V^*||_\infty$, proves that the distance between the policy's value function and the optimal value function converges to zero exponentially. **This completes the demonstration.**

Exercise 2 - Value Iteration Calculation

Given the environment settings, it was asked to compute $V_{k+1}(s_4)$ following the Value Iteration algorithm.

Problem Data:

- Value function at $k = 1$: $V_k = [0, 0, 1, 0, -10]$.
- This means: $V_k(s_1) = 0$, $V_k(s_2) = 0$, $V_k(s_3) = 1$, $V_k(s_4) = 0$, $V_k(s_5) = -10$.
- Dynamics from s_4 : $p(s_4|s_4, a_1) = 0.8$ and $p(s_5|s_4, a_1) = 0.2$.
- Reward function: $r(s, a, s') = 1$ if $s' = s_3$; -10 if $s' = s_5$; 0 otherwise.
- Discount factor: $\gamma = 0.8$.

Derivation:

The Value Iteration update rule is defined by the Bellman optimality operator:

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

For state s_4 , the problem only provides dynamics for action a_1 . So, this is the only action available from s_4 , and the operation \max_a simplifies evaluating the expression for a_1 :

$$V_{k+1}(s_4) = \sum_{s'} p(s'|s_4, a_1) [r(s_4, a_1, s') + \gamma V_k(s')]$$

We expand the summation for the two possible next states, s_4 and s_5 :

$$V_{k+1}(s_4) = p(s_4|s_4, a_1) [r(s_4, a_1, s_4) + \gamma V_k(s_4)] + p(s_5|s_4, a_1) [r(s_4, a_1, s_5) + \gamma V_k(s_5)]$$

Now, we substitute the known values from the problem statement:

- $r(s_4, a_1, s_4) = 0$
- $r(s_4, a_1, s_5) = -10$
- $\gamma = 0.8$
- $p(s_4|s_4, a_1) = 0.8$

- $p(s_5|s_4, a_1) = 0.2$
- $V_k(s_4) = 0$
- $V_k(s_5) = -10$

Substituting these values into the equation:

$$V_{k+1}(s_4) = 0.8 \times (0 + 0.8 \times 0) + 0.2 \times (-10 + 0.8 \times (-10))$$

$$V_{k+1}(s_4) = 0.2 \times [-10 - 8]$$

$$V_{k+1}(s_4) = 0.2 \times (-18)$$

$$V_{k+1}(s_4) = -3.6$$

Practice

Exercise 1: Implementation of Policy Iteration on FrozenLake

The objective of this exercise is to implement the Policy Iteration (PI) algorithm on a modified version of the FrozenLake Gymnasium environment. The algorithm finds an optimal policy by alternating between two phases: Policy Evaluation and Policy Improvement.

Initialization

The implementation begins by initializing the core data structures, including the policy array (π), the value function array ($V(s)$), a state mapping array, and the rewards array.

A key initial step is to map the 2D grid coordinates of the environment to a 1D vector of state indices, which is the standard representation required by the algorithm. The environment is an $N \times N$ grid, where $N = \text{env_size}$. We map each grid coordinate (r, c) (row, column) to a unique state index $k \in \{0, \dots, N^2 - 1\}$. This is accomplished using a row-major traversal, as shown in the code below. This mapping allows us to use k as an index for our 'policy' and 'values' arrays.

```

1 # Enumerate grid states in row-major order
2 k = 0
3 for r in range(env_size):
4     for c in range(env_size):
5         STATES[k] = np.array([r, c], dtype=np.uint8)
6         k += 1

```

Main Algorithm Loop

The Policy Iteration algorithm then enters its main loop. This loop is executed for a maximum number of iterations ('max_iters') as a safeguard, but it is designed to terminate early if the policy becomes stable.

```

1 for i in range(max_iters):
2     # ... Policy Evaluation and Improvement ...

```

Inside this loop, the two core phases are executed sequentially.

Policy Evaluation

The first phase, **Policy Evaluation**, aims to compute the state-value function $V_\pi(s)$ for the current policy π . This is an iterative process that sweeps through all states, applying the Bellman expectation equation as an update rule until the value function converges.

The update rule for a state s under policy π is:

$$V_{k+1}(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s)) [r + \gamma V_k(s')]$$

This inner loop continues until the largest change in the value function across all states (Δ) is smaller than a predefined threshold θ .

```

1 # Policy Evaluation loop
2 while True:
3     delta = 0.0
4     v_old = values.copy()
5
6     for s in range(env.observation_space.n):
7         state = STATES[s]
8
9         # Get the action from the current policy
10        a = policy[s]
11
12        # Get transition probabilities for that action
13        next_state_prob = transition_probabilities(env, state, a, env_size,
14 directions, obstacles).flatten()
15
16        # Terminal (goal) or obstacle cells have no outgoing value
17        done = (state == end_state).all() or obstacles[state[0], state[1]]
18
19        if done:
20            values[s] = 0.0
21        else:
22            # Apply the Bellman expectation update
23            values[s] = (next_state_prob * (REWARDS + gamma * v_old)).sum()
24
25        # Check for convergence
26        delta = max(delta, np.abs(v_old[s] - values[s]))
27
28    if delta < theta:
29        break

```

Policy Improvement

The second phase, **Policy Improvement**, updates the policy π by acting greedily with respect to the value function V_π computed in the previous step.

For each state s , the algorithm finds the action a that maximizes the expected return. This is done by calculating the Q-value, $Q_\pi(s, a)$, for all possible actions from state s and selecting the action that yields the highest Q-value.

The Q-value is defined as:

$$Q_\pi(s, a) = \sum_{s',r} p(s',r|s,a) [r + \gamma V_\pi(s')]$$

The new improved policy π' is then:

$$\pi'(s) \leftarrow \arg \max_a Q_\pi(s, a)$$

```

1 # Policy Improvement step
2 policy_stable = True
3 old_policy = policy.copy()
4

```

```

5 for s in range(env.observation_space.n):
6     state = STATES[s]
7
8     # Store the old action to check for stability
9     b = old_policy[s]
10
11    # Find the best action by looping through all possible actions
12    best_value = -float('inf')
13    best_action = None
14
15    for a in range(env.action_space.n):
16
17        # Get transition probabilities for a specific action 'a'
18        next_state_prob = transition_probabilities(env, state, a, env_size,
19                                                     directions, obstacles).flatten()
20
21        # Calculate the Q-value for this state-action pair
22        va = (next_state_prob * (REWARDS + gamma * values)).sum()
23
24        if va > best_value:
25            best_value = va
26            best_action = a
27
28    # Update the policy with the new best action
29    policy[s] = best_action
30
31    # Check if the policy has changed
32    if best_action != b:
33        policy_stable = False

```

Algorithm Convergence

The main Policy Iteration loop (the outer loop) terminates when the policy is no longer changing. After the Policy Improvement phase, the new policy is compared to the policy from the previous iteration. If no actions have changed for any state, the 'policy_stable' flag remains 'True', and the algorithm breaks, having found the optimal policy π^* .

```

1 # Check for convergence of the outer loop
2 if policy_stable:
3     break

```

Exercise 2: Implementation of the iLQR Algorithm on Pendulum

The task is to implement the Iterative Linear Quadratic Regulator (iLQR) algorithm for the Pendulum-v1 environment. The work, is divided into three main components: complete the implementation of the pendulum's dynamics model, the iLQR backward pass, and the iLQR forward pass.

Pendulum Dynamics Model

The `pendulum_dyn` function implements the discrete-time dynamics of the pendulum, as specified in the assignment instructions. Given the current state $x = [\theta, \dot{\theta}]$ and control u , it computes the next state $x_{t+1} = [\theta_{t+1}, \dot{\theta}_{t+1}]$.

The code implements the following equations of motion:

$$\begin{aligned}\dot{\theta}_{t+1} &= \dot{\theta}_t + \left(\frac{3g}{2l} \sin(\theta_t) + \frac{3.0}{ml^2} u_t \right) dt \\ \theta_{t+1} &= \theta_t + \dot{\theta}_{t+1} dt\end{aligned}$$

The state is then returned as $x_{t+1} = [\theta_{t+1}, \dot{\theta}_{t+1}]$.

```

1     # TODO
2     newthdot = thdot + ((3 * g) / (2 * l) * np.sin(th) + 3.0 / (m * l**2) * u) * dt
3     newth = th + newthdot * dt

```

iLQR Backward Pass

The `backward` function is the core of the iLQR algorithm. It works by stepping backward in time, starting from the final goal (at time $t = H - 1$) and moving all the way to the beginning (at time $t = 0$).

As it moves backward, it figures out the best way to update the controls. It calculates four key values at each time step: k_t , K_t , p_t , and P_t .

- p_t and P_t : These terms describe the "cost-to-go," which is the expected total cost from the current time t until the end. They are calculated at the final step and passed backward in time.
- k_t and K_t : These terms are the components of the new, updated linear control policy.

The code implements the formulas for these terms, including the ones for k_t and for p_t provided in the assignment sheet.

```

1     # TODO
2     kt = -np.linalg.inv(Rt + Bt.T @ Pt1 @ Bt) @ (rt + Bt.T @ pt1)
3     Kt = -np.linalg.inv(Rt + Bt.T @ Pt1 @ Bt) @ Bt.T @ Pt1 @ At
4     # TODO
5     pt = qt + Kt.T @ (Rt @ kt + rt) + (At + Bt @ Kt).T @ pt1 + (At + Bt @ Kt)
6     .T @ Pt1 @ Bt @ kt
       Pt = (Qt + Kt.T @ Rt @ Kt + (At + Bt @ Kt).T @ Pt1 @ (At + Bt @ Kt))

```

iLQR Forward Pass

The `forward` function simulates the system's dynamics forward in time using the new control policy computed during the backward pass. It applies the control update using the calculated gain matrices K_t and k_t , as specified by the formula:

$$\text{control} = k_t + K_t(x_t - \bar{x}_t)$$

In the code, \bar{x}_t is the nominal trajectory (`x_seq[t]`) and x_t is the new simulated trajectory (`x_seq_hat[t]`). The final control is then clipped to the environment's action-space limits.

```

1     # TODO
2     control = k_seq[t] + K_seq[t] @ (x_seq_hat[t] - x_seq[t])
3
4     # clip controls to the actual range from gymnasium
5     u_seq_hat[t] = np.clip(u_seq[t] + control, -2, 2)

```

Bibliography

References

- [1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second Edition, MIT Press, 2018. (Specifically Chapter 4).
- [2] Jiantao Jiao. "Lecture 17: Bellman Operators, Policy Iteration, and Value Iteration". *EE 290 Theory of Multi-armed Bandits and Reinforcement Learning*, UC Berkeley, 2021.
- [3] Akshay Krishnamurthy. "Lecture 6: Policy improvement methods". Course notes.
- [4] *Lecture 4: Planning in MDPs*. RL Theory course notes. Available at: <https://rltheory.github.io/lecture-notes/planning-in-mdps/lec4/>
- [5] Robotic Exploration Lab, Carnegie Mellon University. *iLQR Tutorial*. Available at: https://rexlab.ri.cmu.edu/papers/iLQR_Tutorial.pdf
- [6] Jonathan Hui. "RL — LQR & iLQR Linear Quadratic Regulator". *Towards Data Science*, Medium, 2018. Available at: <https://jonathan-hui.medium.com/rl-lqr-ilqr-linear-quadratic-regulator-a5de5104c750>