Online and Reinforcement Learning (2025)
Home Assignment 7

Davide Marchi 777881

Contents

1 Short Questions

1. **True.** *Justification:* In a finite average-reward MDP with a finite diameter the MDP is communicating (in fact, ergodic if every state is reachable from every other under some policy). This implies that the long-run average reward (gain) is independent of the starting state.

2. **False.** *Justification:* Finite diameter guarantees that there exists a policy which can reach any state from any other in a finite expected number of steps. However, this does not mean that every arbitrary choice of actions will eventually reach every state.

3. **False.** *Justification:* In an ergodic MDP the optimal gain is unique (i.e., independent of the initial state), but the optimal bias function is determined only up to an additive constant. Hence, the bias is not uniquely defined in an absolute sense.

4. **False.** *Justification:* A PAC-MDP algorithm guarantees that the number of ε-bad (i.e., non-ε-optimal) steps is bounded with high probability. However, it does not imply that there exists a finite time after which every subsequent policy is ε-optimal; occasional exploration may still yield suboptimal actions.

2 Offline Evaluation of Bandit Algorithms - the Practical Part

2.1 Alghorithms

Modified UCB1 for Offline Evaluation

In this subsection, we present a modified version of the UCB1 algorithm for offline evaluation under a uniform logging policy. Given a log of T rounds in which, at each round t, the logging policy selects an arm A_t uniformly at random and observes a loss $\ell_{A_t,t} \in [0,1]$, we replay UCB1. Updates occur only when the arm chosen by UCB1 matches the logged arm; in such cases, the observed loss is scaled by the importance weight K to yield an unbiased estimate.

Offline UCB1 with Importance Weighting:

1. **Input:** Number of arms K, total rounds T, logged data $\{(A_t, \ell_{A_t,t})\}_{t=1}^{T}$, constant $c > 0$.

2. **Initialize:** For each arm $i = 1, \ldots, K$, set $L_i \leftarrow 0$ and $N_i \leftarrow 0$.

3. **For** $t = 1$ to T:

   (a) For $i = 1$ to K:

   - If $N_i = 0$, set $UCB_i \leftarrow +\infty$.

- Otherwise, set

$$UCB_i \leftarrow \frac{L_i}{N_i} + c\sqrt{\frac{\ln(t)}{N_i}}.$$

(b) Let $i_t \leftarrow \arg\min_i UCB_i$.

(c) **If** $A_t = i_t$, then update

$$L_{A_t} \leftarrow L_{A_t} + K\,\ell_{A_t,t}, \quad N_{A_t} \leftarrow N_{A_t} + 1.$$

4. **Output:** The pairs $\{(L_i,\, N_i)\}_{i=1}^K$.

Modified EXP3 for Offline Evaluation (Anytime Version)

Here, we describe a modified version of the EXP3 algorithm adapted for offline evaluation in the anytime setting. With a uniform logging policy, the logged data is used to update the cumulative importance-weighted loss only when the logging policy's chosen arm coincides with the arm that the EXP3 algorithm would have drawn. In the anytime version, the learning rate is set to vary with time as

$$\eta_t = \sqrt{\frac{2\ln(K)}{K\,t}}.$$

Offline EXP3 with Importance Weighting:

1. **Input:** Number of arms K, total rounds T, logged data $\{(A_t, \ell_{A_t,t})\}_{t=1}^T$.
2. **Initialize:** For each $i = 1, \ldots, K$, set $w_i \leftarrow 1$ and $\tilde{L}_i \leftarrow 0$.
3. **For** $t = 1$ to T:

(a) Set

$$\eta_t \leftarrow \sqrt{\frac{2\,\ln(K)}{K\,t}}.$$

(b) Compute

$$W \leftarrow \sum_{i=1}^{K} \exp\bigl(-\eta_t\,\tilde{L}_i\bigr).$$

(c) For each arm i, set

$$p_t(i) \leftarrow \frac{\exp(-\eta_t\,\tilde{L}_i)}{W}.$$

(d) **If** A_t is the arm drawn according to p_t, then update

$$\tilde{L}_{A_t} \leftarrow \tilde{L}_{A_t} + K\,\ell_{A_t,t}.$$

4. **Output:** The final cumulative losses $\{\tilde{L}_i\}_{i=1}^K$ and the distribution $\{p_T(i)\}_{i=1}^K$.

2.1.1 Practical Part

In this part, I implement the offline versions of UCB1 and EXP3 with importance-weighted data under a uniform logging policy. The following code snippets show the essential steps:

- **Offline UCB1**: I update each arm's statistics only when the algorithm's chosen arm matches the logging policy's arm. The observed reward is scaled by K (the number of arms).
- **Offline EXP3 (Anytime)**: I use $\eta_t = \sqrt{\frac{2\ln(K)}{K\,t}}$ and likewise update only on rounds where the chosen arm matches the logged arm, again scaling the reward by K.
- **Fixed-arm baselines**: To compare, I also evaluate single-arm (and subsets of arms) policies offline, accumulating $K \cdot R_t$ whenever the logged arm is in the chosen subset.

```
def offline_ucb1(A, R, K, c=2.0):
    T = len(A)
    L = np.zeros(K)              # sums of K * reward
    N = np.zeros(K, dtype=int) # counts of updates
    cum_rewards = np.zeros(T)
    for t in range(T):
        # compute UCB indices
        ucb_vals = np.array([
            (L[i]/N[i] + c*np.sqrt(np.log(t+1)/N[i])) if N[i]>0 else np.
   inf
            for i in range(K)
        ])
        it = np.argmin(ucb_vals)
        # update if match
        if A[t] == it:
            iw_reward = K * R[t]
            L[it] += iw_reward
            N[it] += 1
            cum_rewards[t] = (cum_rewards[t-1] + iw_reward) if t>0 else
   iw_reward
        else:
            cum_rewards[t] = cum_rewards[t-1] if t>0 else 0.0
    return cum_rewards

def offline_exp3(A, R, K):
    T = len(A)
    Ltilde = np.zeros(K)
    cum_rewards = np.zeros(T)
    for t in range(T):
        eta_t = np.sqrt((2.0*np.log(K)) / (K*(t+1)))
        w = np.exp(-eta_t * Ltilde)
        p = w / np.sum(w)
        # update with importance weighting if match
        chosen_arm = A[t]
```

```
        iw_reward = K * R[t]
        Ltilde[chosen_arm] += K * R[t]  # or K*(1 - R[t]) if using loss=1
    -reward
        cum_rewards[t] = (cum_rewards[t-1] + iw_reward) if t>0 else
    iw_reward
    return cum_rewards

def offline_fixed_subset(A, R, subset_of_arms, K):
    T = len(A)
    cumr = np.zeros(T)
    subset = set(subset_of_arms)
    for t in range(T):
        iw_reward = K*R[t] if A[t] in subset else 0.0
        cumr[t] = (cumr[t-1] + iw_reward) if t>0 else iw_reward
    return cumr
```

Listing 1: Key implementation of Offline UCB1 and Offline EXP3 with importance weighting.

After running these algorithms on the logged dataset, I produce five plots as requested:

- (a) Cumulative reward for *all arms* (single-arm baselines) in faint lines, plus UCB1, EXP3, and an optional EXP3 bound.
- (b) Comparison of the *best* and *worst* arms vs. UCB1 and EXP3.
- (c) *Best* arm vs. the *two worst* arms, plus UCB1 and EXP3.
- (d) *Best* arm vs. the *three worst* arms, plus UCB1 and EXP3.
- (e) *Best*, *median*, and *worst* arms, plus UCB1 and EXP3.

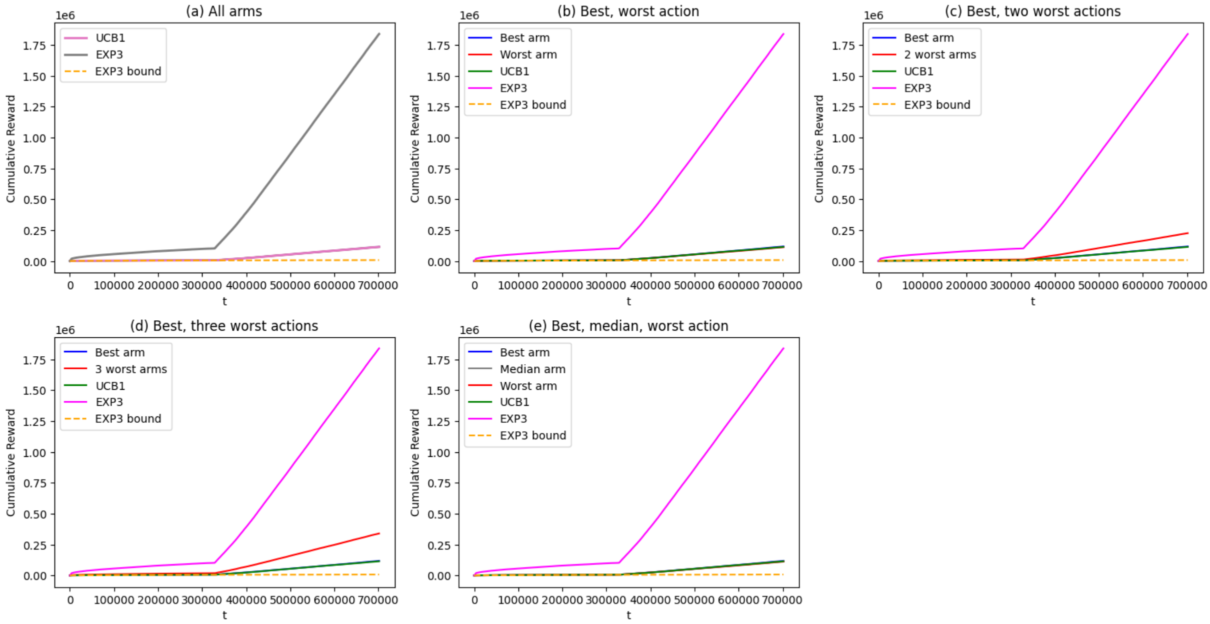

Figure 1: Offline evaluation results: UCB1 and EXP3 compared with fixed-arm baselines.

These plots confirm that the offline EXP3 and UCB1 algorithms achieve high cumulative reward relative to the worst arms, and in many cases approach or surpass the best single-arm baseline, illustrating their effectiveness in this offline replay setting.

3 Grid-World: Continual and undiscounted

In this question, we model the 4-room grid-world as an average-reward MDP and solve it using Value Iteration (VI).

(i) Implementation of Value Iteration for Average-Reward MDPs

The following Python function implements VI for average-reward MDPs. The algorithm computes an approximation of the optimal gain g^*, the bias function b^* (normalized such that $\min_s b^*(s) = 0$), and the optimal policy.

```
def average_reward_vi(mdp, epsilon=1e-6, max_iter=10000):

    # Initialize the value function arbitrarily (here, zeros)
    V = np.zeros(mdp.nS)

    for iteration in range(max_iter):
        V_next = np.zeros(mdp.nS)
        # Update V_next for each state s using the Bellman operator:
        # V_next(s) = max_a [ R(s, a) + sum_x P(x|s,a) * V(x) ]
        for s in range(mdp.nS):
```

```
             action_values = []
             for a in range(mdp.nA):
                 q_sa = mdp.R[s, a] + np.dot(mdp.P[s, a], V)
                 action_values.append(q_sa)
             V_next[s] = max(action_values)

         # Compute the difference (increment) vector
         diff = V_next - V
         # The span (max difference minus min difference) is our stopping
criterion
         span_diff = np.max(diff) - np.min(diff)
         V = V_next.copy()

         if span_diff < epsilon:
             print(f"Converged after {iteration+1} iterations with span {
span_diff:.2e}.")
             break
     else:
         print("Warning: Maximum iterations reached without full
convergence.")

     # Estimate the optimal gain g* as the average of the maximum and
minimum differences.
     gain = 0.5 * (np.max(diff) + np.min(diff))
     # The bias function is approximated by normalizing V (bias is defined
 up to an additive constant)
     bias = V - np.min(V)

     # Derive the optimal policy: for each state, choose the action
maximizing:
     # Q(s, a) = R(s, a) + sum_x P(x|s,a) * V(x)
     policy = np.zeros(mdp.nS, dtype=int)
     for s in range(mdp.nS):
         best_val = -np.inf
         best_a = 0
         for a in range(mdp.nA):
             q_sa = mdp.R[s, a] + np.dot(mdp.P[s, a], V)
             if q_sa > best_val:
                 best_val = q_sa
                 best_a = a
         policy[s] = best_a

     return policy, gain, bias
```
Listing 2: Average-Reward Value Iteration Function

(ii) Visualization of the Optimal Policy

Running the above function on the grid-world produced the following output:

- **Optimal Gain:** $g^* = 0.07563$

- **Span of the Optimal Bias:** $\mathrm{sp}(b^*) = 0.92437$

The optimal policy for the grid-world is represented by the following table. The arrows indicate the action chosen in each state (with walls marked as "Wall"):

Wall	Wall	Wall	Wall	Wall	Wall	Wall
Wall	→	→	→	↓	↓	Wall
Wall	↓	↑	Wall	↓	←	Wall
Wall	↓	Wall	Wall	↓	Wall	Wall
Wall	→	→	→	→	↓	Wall
Wall	→	↑	Wall	→	↑	Wall
Wall	Wall	Wall	Wall	Wall	Wall	Wall

Table 1: Optimal Policy Grid: Arrows indicate the optimal actions.

(iii) Interpretation of $1/g^*$

In this grid-world, the optimal gain g^* represents the long-run average reward per time step. Hence, the quantity

$$\frac{1}{g^*} \approx \frac{1}{0.07563} \approx 13.22$$

can be interpreted as the average number of steps needed to collect one unit of reward under the optimal policy. In other words, the agent gathers one reward approximately every 13 time steps, reflecting the efficiency of the optimal strategy in this continual setting.

4 An Empirical Evaluation of UCB Q-learning

In this exercise we implement UCB Q-learning in the 5-state RiverSwim environment. The algorithm is run with the parameters:

$$\gamma = 0.92, \quad \varepsilon = 0.13, \quad \delta = 0.05, \quad T = 2 \times 10^6.$$

The starting state is sampled uniformly and we set

$$H = \left\lceil \frac{1}{1-\gamma} \log \frac{1}{\varepsilon} \right\rceil, \quad b(k) = \sqrt{\frac{H}{k} \log \Big(SA \log(k+1)/\delta \Big)}.$$

The performance is measured by the cumulative number of ε-bad steps,

$$n(t) = \sum_{\tau=1}^{t} \mathbf{1}\{V^{\pi_\tau}(s_\tau) < V^*(s_\tau) - \varepsilon\},$$

where $V^*(s)$ is computed via value iteration. In our experiments the value iteration procedure yields:

$$V^* = [2.83439639,\ 3.45056954,\ 4.27771502,\ 5.31104628,\ 6.594788].$$

Below is the complete implementation of the UCB Q-learning routine.

```python
def run_ucb_ql(T, gamma, epsilon_eval, delta, H, V_star, record_interval=
    1000, seed=None):
    if seed is not None:
        np.random.seed(seed)
    env = riverswim(5)
    # Sample starting state uniformly.
    env.s = np.random.randint(0, env.nS)
    s = env.s
    S = env.nS
    A = env.nA
    R_max = 1.0
    optimistic_init = R_max / (1.0 - gamma)
    Q = np.full((S, A), optimistic_init)
    Q_hat = np.copy(Q)
    N = np.ones((S, A))
    n_bad = 0
    times = []
    n_values = []
    # Cache for policy evaluation
    policy_value_cache = {}
    for t in range(1, T + 1):
        # Select action greedily w.r.t. Q_hat.
        a = np.argmax(Q_hat[s, :])
        new_s, r = env.step(a)
        k = N[s, a]
        alpha_k = (H + 1) / (H + k)
        bonus = np.sqrt((H / k) * np.log(S * A * np.log(k + 1) / delta))
        # Q-update with exploration bonus.
        Q[s, a] = (1 - alpha_k) * Q[s, a] + alpha_k * (r + bonus + gamma
    * np.max(Q_hat[new_s, :]))
        Q_hat[s, a] = min(Q_hat[s, a], Q[s, a])
        N[s, a] += 1
        # Evaluate current greedy policy.
        policy = np.argmax(Q_hat, axis=1)
        policy_tuple = tuple(policy)
        if policy_tuple not in policy_value_cache:
            V_policy = evaluate_policy(policy, env, gamma)
            policy_value_cache[policy_tuple] = V_policy
        else:
            V_policy = policy_value_cache[policy_tuple]
        if V_policy[new_s] < V_star[new_s] - epsilon_eval:
            n_bad += 1
        s = new_s
        if t % record_interval == 0:
            times.append(t)
            n_values.append(n_bad)
    return np.array(times), np.array(n_values)
```

Listing 3: Complete UCB Q-learning routine

(i) Sample Path of $n(t)$

A single run of the algorithm produces a sample path of the cumulative ε-bad steps $n(t)$. Figure 2 shows the sample path. This plot illustrates the steady accumulation of bad steps over time.

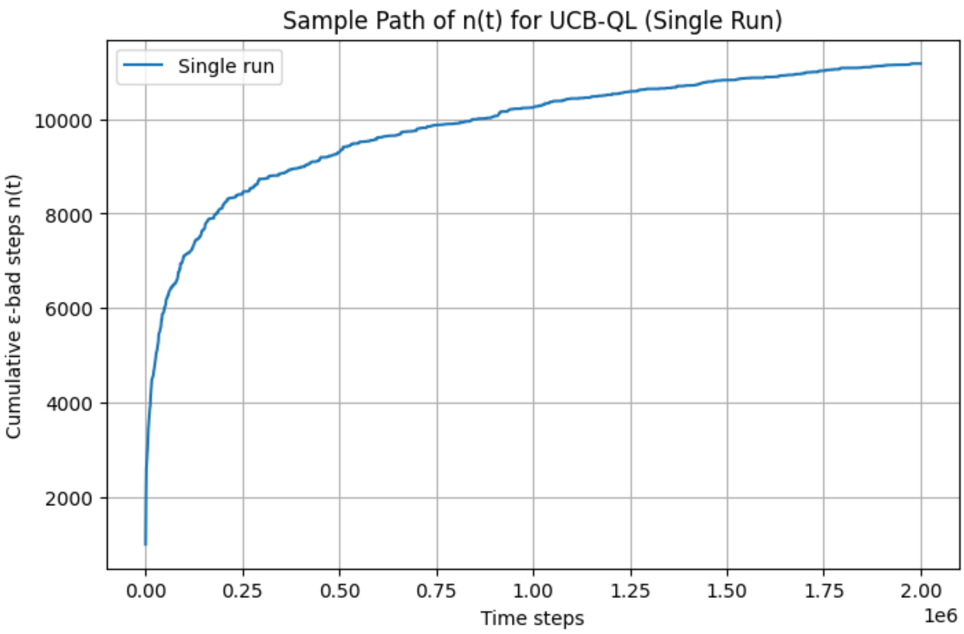

Figure 2: Sample path of the cumulative number of ε-bad steps $n(t)$ for a single run.

(ii) Averaged $n(t)$ over 100 Runs

In addition, the algorithm was executed over 100 independent runs. Figure 3 shows the average $n(t)$ with 95% confidence intervals. Notably, the individual runs exhibit very similar behavior, as evidenced by the tight confidence bounds.

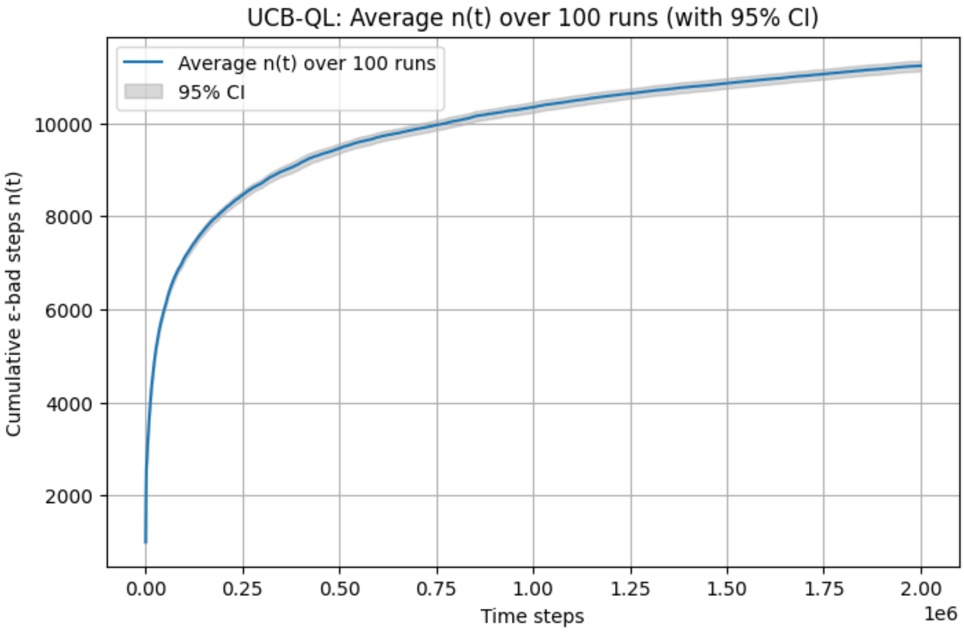

Figure 3: Averaged $n(t)$ over 100 runs with 95% confidence intervals.

In conclusion, the experimental results confirm that UCB Q-learning consistently maintains a similar behavior across runs, with the cumulative ε-bad steps growing as expected in the RiverSwim environment.