Online and Reinforcement Learning (2025)
Home Assignment 5

Davide Marchi 777881

Contents

1 Deep Q-Learning

1.1 Neural architecture

1.1.1 Structure

The line

```
x = torch.cat((x_input, x), dim=1)
```

takes the original input `x_input` (the raw state features) and concatenates it with the hidden representation `x` along the feature dimension. This effectively merges the initial state features with the learned features from the hidden layers. Such a design is sometimes referred to as a *skip connection* or *residual-like* connection, because the network has direct access to the original input when producing the final Q-values.

1.1.2 Activation function

We use tanh instead of the standard logistic sigmoid function because:

- tanh is zero-centered (ranging from -1 to 1), which often helps with training stability.
- The logistic (sigmoid) function saturates more easily at 0 or 1, leading to slower gradients. In contrast, tanh keeps activations in a range that often accelerates convergence in practice.

1.2 Adding the Q-learning

The missing line to compute the target **y** (right after the comment "`# Compute targets`") is shown below. We also detach the tensor so that the gradient does not flow back through the next-state values:

```
max_elements = torch.max(target_Qs, dim=1)[0].detach()
y = rewards + gamma * max_elements
```

This implements the standard Q-learning target:

$$y_i = r_i + \gamma \max_{a'} Q(s_i', a').$$

After adding this line, the training converged to policies achieving returns above 200 in many episodes, indicating that the agent successfully learned to land.

Learning curve

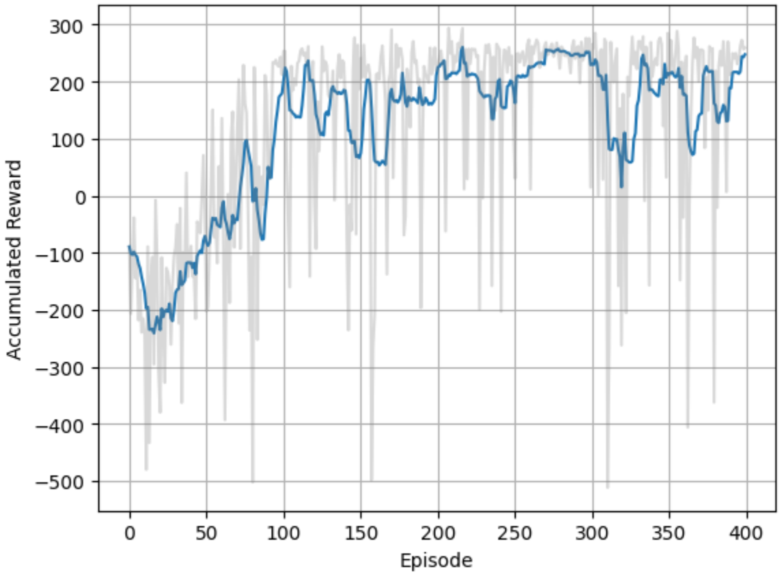

Figure 1: Accumulated reward per episode (in gray) and its smoothed average (blue).

1.3 Epsilon

The line

```
 explore_p = explore_stop + (explore_start - explore_stop)*np.exp(-
decay_rate*ep)
```

implements an exponentially decaying exploration probability. At the beginning (when `ep` $= 0$), it starts near `explore_start`, and as the episode index `ep` grows, the probability `explore_p` exponentially approaches `explore_stop`. This ensures that early in training the agent explores more, and then it gradually exploits its learned policy more often.

1.4 Gather

The line

```
 Q_tensor = torch.gather(output_tensor, 1, actions_tensor.unsqueeze(-
1)).squeeze()
```

selects the Q-value corresponding to the action actually taken in each state. Since `output_tensor` contains Q-values for all possible actions, we use `gather` along the action dimension to pick out the one Q-value that corresponds to the `actions_tensor`. In other words, it is a convenient way to index a batch of Q-value vectors by their chosen actions.

1.5 Target network

Code modifications

Below is the code I introduced to maintain a target network and perform Polyak averaging. First, I created `targetQN` right after creating `mainQN` and copied the parameters:

```
# Create the target network
targetQN = QNetwork(hidden_size=hidden_size)

# Copy parameters from mainQN to targetQN so they start identical
targetQN.load_state_dict(mainQN.state_dict())
```

Then, at the end of each training step (i.e., right after `optimizer.step()`), I inserted the Polyak update:

```
with torch.no_grad():
    for target_param, main_param in zip(targetQN.parameters(), mainQN.
    parameters()):
        target_param.data.copy_(tau * main_param.data + (1.0 - tau) *
    target_param.data)
```

This implements

$$\theta_{\text{target}} \leftarrow \tau\,\theta_{\text{main}} \;+\; (1-\tau)\,\theta_{\text{target}},$$

where $\tau \in (0, 1)$ is the blend factor.

The changes corrctly introduce a delayed target Q-network. We copy the main network's parameters once at initialization, then repeatedly blend them after every gradient update. This ensures that the target network changes slowly, providing more stable target values in the Q-learning loss.

Comparison with and without the target network

I kept the same hyperparameters as before and introduced the target network. The learning curve is shown below.

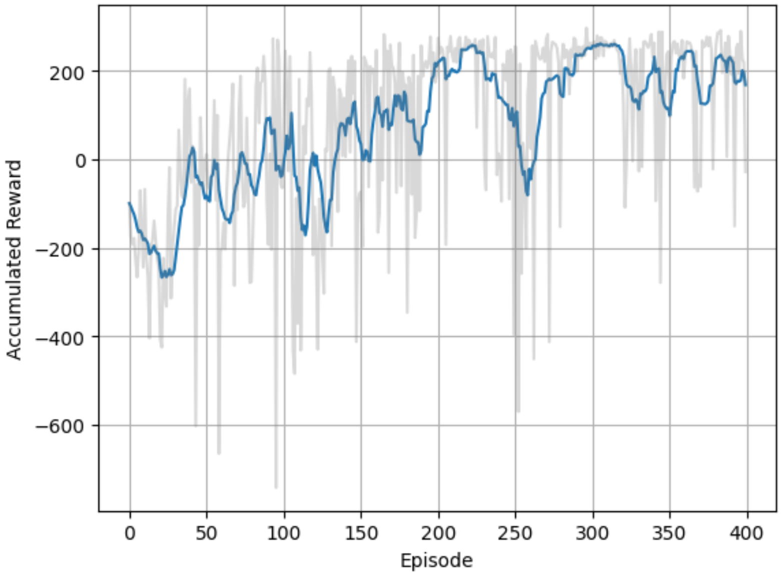

Figure 2: Accumulated reward per episode (in gray) and its smoothed average (blue) using a target network.

Empirically, we can see that the performance still reaches high returns, and in many cases the learning appears somewhat more stable. Although there can be noise and variability in individual runs, a target network typically reduces training instabilities and leads to more consistent convergence.

2 A tighter analysis of the Hedge algorithm

1(a). Apply Hoeffding's Lemma in the Hedge Analysis.

We consider the Hedge algorithm, where at each round $t = 1, \dots, T$:

- We have K experts and maintain a cumulative loss $L_{t-1}(a)$ for each expert a.
- We form a distribution

$$p_t(a) \;=\; \frac{\exp\bigl(-\eta\, L_{t-1}(a)\bigr)}{\sum_{a'} \exp\bigl(-\eta\, L_{t-1}(a')\bigr)}.$$

- We then observe the losses $\{\ell_t(a)\}_{a=1}^{K}$ (full information), and update $L_t(a) = L_{t-1}(a) + \ell_t(a)$.

Define

$$W_t \;=\; \sum_{a=1}^{K} \exp\bigl(-\eta\, L_t(a)\bigr).$$

Initially, $W_0 = K$ because $L_0(a) = 0$ for all a.

To bound the regret, we look at

$$\sum_a p_t(a)\, e^{-\eta\, \ell_t(a)} \;=\; \mathbb{E}_{A\sim p_t}\Big[e^{-\eta\, \ell_t(A)}\Big].$$

Here, $\ell_t(A) \in [0,1]$ (assuming losses are bounded in $[0,1]$). We apply Hoeffding's Lemma with $\alpha = -\eta < 0$. Then:

$$\ln\Big(\mathbb{E}\big[e^{-\eta\, \ell_t(A)}\big]\Big) \;\le\; -\eta\, \mathbb{E}\big[\ell_t(A)\big] \;+\; \tfrac{\eta^2}{8}.$$

Since $\mathbb{E}[\ell_t(A)] = \sum_a p_t(a)\, \ell_t(a)$, we get

$$\ln\Big(\sum_a p_t(a)\, e^{-\eta\, \ell_t(a)}\Big) \;\le\; -\eta \sum_a p_t(a)\, \ell_t(a) \;+\; \frac{\eta^2}{8}.$$

1(b). Find the η that Minimizes the New Bound.

Summing the above inequality over $t = 1, \ldots, T$ yields:

$$\sum_{t=1}^{T} \ln\Big(\sum_a p_t(a)\, e^{-\eta\, \ell_t(a)}\Big) \;\le\; -\eta \sum_{t=1}^{T} \sum_a p_t(a)\, \ell_t(a) \;+\; \frac{\eta^2\, T}{8}.$$

On the other hand, from the definition of W_T we have

$$\frac{W_T}{W_0} \;=\; \prod_{t=1}^{T} \sum_a p_t(a)\, e^{-\eta\, \ell_t(a)} \;\ge\; \frac{1}{K}\, \exp\Big(-\eta\, \min_a L_T(a)\Big).$$

Taking the logarithm and rearranging gives:

$$-\frac{1}{\eta} \sum_{t=1}^{T} \ln\Big(\sum_a p_t(a)\, e^{-\eta\, \ell_t(a)}\Big) \;\ge\; \min_a L_T(a) \;-\; \frac{\ln(K)}{\eta}.$$

Combine this with the previous sum to bound the *regret*:

$$\sum_{t=1}^{T} \sum_a p_t(a)\, \ell_t(a) \;-\; \min_a L_T(a) \;\le\; \frac{\ln(K)}{\eta} \;+\; \frac{\eta\, T}{8}.$$

Denote the regret by R_T. We see

$$R_T \;\le\; \frac{\ln(K)}{\eta} \;+\; \frac{\eta\, T}{8}.$$

To minimize the right-hand side, we set

$$\frac{d}{d\eta}\Big(\tfrac{\ln(K)}{\eta} + \tfrac{\eta\, T}{8}\Big) \;=\; -\frac{\ln(K)}{\eta^2} + \frac{T}{8} \;=\; 0 \quad\Longrightarrow\quad \eta \;=\; \sqrt{\frac{8\, \ln(K)}{T}}.$$

1(c). Final Regret Bound.

Plugging $\eta = \sqrt{\frac{8\ln(K)}{T}}$ back into the bound, we get

$$R_T \;\leq\; \ln(K)\sqrt{\frac{T}{8\ln(K)}} \;+\; \sqrt{\frac{8\ln(K)}{T}}\,\frac{T}{8} \;=\; 2\sqrt{\frac{T\ln(K)}{8}} \;=\; \sqrt{\frac{T\ln(K)}{2}}.$$

So the regret of Hedge can be bounded by

$$\boxed{R_T \;\leq\; \sqrt{\frac{T\ln(K)}{2}}\,.}$$

This shows a sharper constant than the usual bound $O(\sqrt{T\ln K})$, thanks to applying Hoeffding's Lemma directly.

2. Why the Same Approach Cannot Be Used to Tighten the Regret Bound for EXP3.

In the EXP3 algorithm, which deals with *bandit* feedback, we do *not* get to observe the losses $\ell_t(a)$ of all experts a at each round. Instead, we only see the loss of the action we actually chose. Therefore, to update our weights, we rely on an *unbiased but potentially high-variance estimator* of $\ell_t(a)$ for the unchosen actions.

Hoeffding's Lemma in the form we used above requires direct access to the true $\ell_t(a)$ for the expectation inside the exponential. In EXP3, the random variable inside the exponential is replaced by the *estimated* loss, which typically has higher variance and does not stay nicely within $[0,1]$ without additional complications. Consequently, we cannot directly apply the same sub-Gaussian/Hoeffding-type bound to get the improved constant. The partial information setting forces us to handle variance in the importance-weighted estimators, and that prevents us from achieving the same neat constant factor that we get in the full-information Hedge analysis.

3 Empirical evaluation of algorithms for adversarial environments

In adversarial settings, an algorithm's performance is evaluated under the assumption that an adversary will create the most difficult possible scenarios. While we can try to test algorithms experimentally, it's impossible to cover every possible adversarial strategy.

One key issue is that showing an algorithm is not robust is relatively easy. If we can find or construct an input where the algorithm fails, that's enough to prove it has weaknesses. However, proving that an algorithm is robust is much harder. This would require showing that the algorithm works against every possible attack, but the space of adversarial strategies is often too large to test exhaustively.

A few challenges make a full evaluation difficult:

- **Unlimited adversarial strategies:** Adversaries can design attacks based on how the algorithm works, and there's no way to test for all possible attacks.
- **Adversaries can adapt over time:** Especially in real-world scenarios attackers constantly change their strategies, meaning any static test will only capture a limited picture.
- **Resource constraints:** Running large-scale experiments that simulate adversarial behavior is often too expensive and time-consuming to be practical.

Conclusion

Since the space of possible adversarial attacks is too large and adaptive, no set of experiments can fully prove an algorithm is robust. Empirical testing can reveal weaknesses, but it can never provide a guarantee that an algorithm will hold up against every possible adversarial scenario.

4 Empirical comparison of UCB1 and EXP3 algorithms

In this exercise, I analyze the performance of two bandit algorithms, a modified version of UCB1 and EXP3, under two different scenarios. First, I compare their performance in a standard i.i.d. environment where the best arm has a bias of $\mu^* = 0.5$ and the suboptimal arms have mean $\mu^* - \Delta$, with $\Delta \in \{0.25, 0.125, 0.0625\}$, and the number of arms $K \in \{2, 4, 8, 16\}$. Second, I design a worst-case adversarial scenario to "break" UCB1, showing that even though UCB is robust in many cases, it can be forced to perform poorly under a specifically crafted reward sequence. In contrast, EXP3 (designed for adversarial settings) adapts much better.

Normal Comparison (IID Setting)

In the i.i.d. experiments the modified UCB1 algorithm is used with the bonus term:

$$\text{bonus} = \sqrt{\frac{\log t}{n_a}},$$

where n_a is the number of times arm a has been played and t is the current time step. The EXP3 algorithm uses a time-varying learning rate

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

Below is the code snippet for the EXP3 algorithm that was used:

```python
def run_exp3(T, means, nonstationary=False, worst_case=False, epsilon=
   0.01):
    n_arms = len(means)
    best_mean = np.max(means)  # Note: This may lead to negative regret
   initially
    regrets = np.zeros(T)
    cumulative_regret = 0.0
    L = np.zeros(n_arms)  # cumulative losses for each arm

    for t in range(T):
        eta_t = np.sqrt(np.log(n_arms) / ((t+1) * n_arms))
        L_min = np.min(L)
        w = np.exp(-eta_t * (L - L_min))
        p = w / np.sum(w)
        chosen_arm = np.random.choice(n_arms, p=p)

        if nonstationary and worst_case:
            reward = adversarial_reward(t, chosen_arm, T, epsilon)
        elif nonstationary:
            if chosen_arm == 1 and t >= T/2:
                reward = bernoulli_sample(0.0)
            else:
                reward = bernoulli_sample(means[chosen_arm])
        else:
            reward = bernoulli_sample(means[chosen_arm])

        cumulative_regret += (best_mean - reward)
        regrets[t] = cumulative_regret

        loss = (1 - reward) / max(p[chosen_arm], 1e-12)
        L[chosen_arm] += loss

    return regrets
```

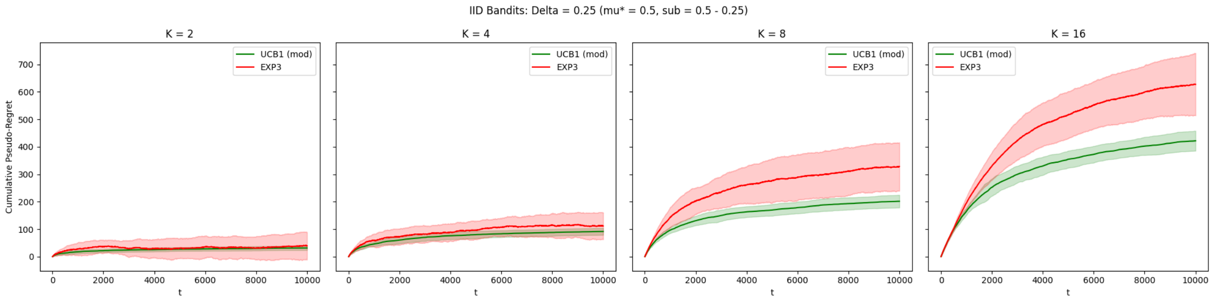

Figure 3: Cumulative pseudo-regret for $\Delta = 0.25$, best arm $\mu^* = 0.5$, suboptimal arms $= 0.5 - 0.25$. We compare UCB1 (modified) and EXP3 over $K = 2, 4, 8, 16$.

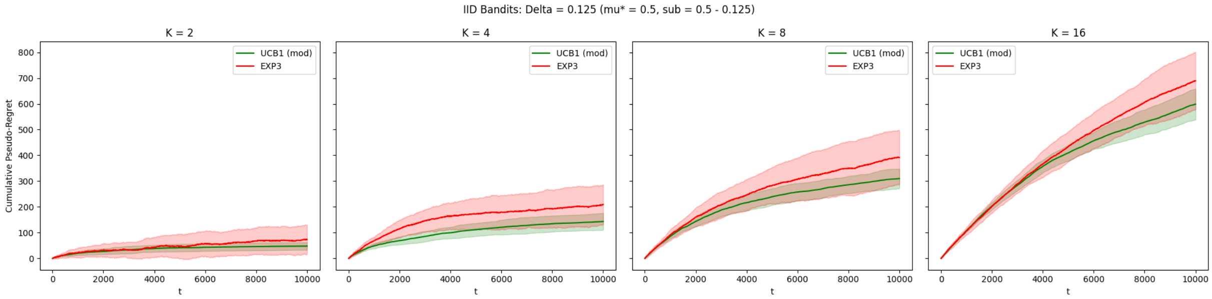

Figure 4: Cumulative pseudo-regret for $\Delta = 0.125$, best arm $\mu^* = 0.5$, suboptimal arms $= 0.5 - 0.125$. We compare UCB1 (modified) and EXP3 over $K = 2, 4, 8, 16$.

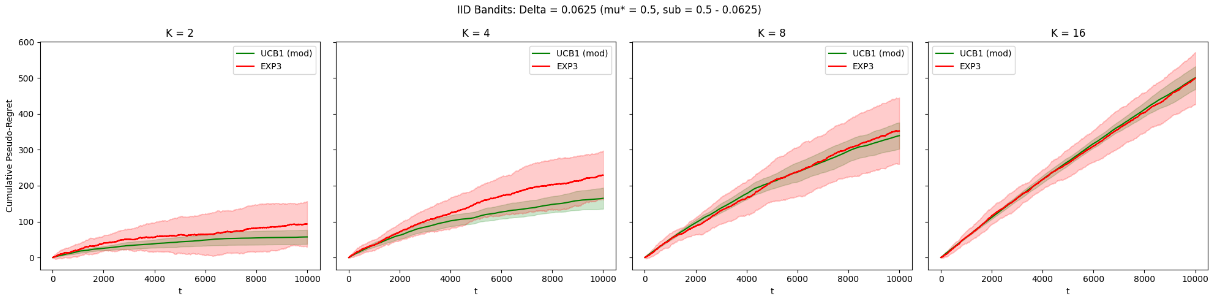

Figure 5: Cumulative pseudo-regret for $\Delta = 0.0625$, best arm $\mu^* = 0.5$, suboptimal arms $= 0.5 - 0.0625$. We compare UCB1 (modified) and EXP3 over $K = 2, 4, 8, 16$.

Discussion:

The plots above illustrate the cumulative pseudo-regret for both algorithms in the i.i.d. setting. It can be observed that:

- The modified UCB1 algorithm performs overall better in the i.i.d. environment, and this advantage is especially noticeable for large K and large Δ.
- EXP3 sometimes starts with negative regret because its pseudo-regret is computed as (best_mean − reward), and early on the reward can exceed the nominal best mean.
- The variance (or standard deviation) of EXP3's regret is often larger, as indicated by the wider confidence bands, reflecting more variability in its performance across runs.

Worst-Case Adversarial Setting (Breaking UCB)

To force UCB1 to perform poorly, I designed a worst-case adversarial scenario. The adversarial reward schedule is defined as follows:

- For $t < T/2$:
  - Arm 0: reward $= 0$.
  - Arm 1: reward $= 1$.
- For $t \geq T/2$:
  - Arm 0: reward $= 1$.
  - Arm 1: reward $= 1$ with probability ϵ, and 0 otherwise.

This reward structure is chosen so that UCB1, which initially observes high rewards for arm 1, builds up a high empirical average and large sample count on that arm. When the reward pattern switches, UCB1 continues to favor arm 1 and thus suffers nearly linear regret. Meanwhile, EXP3 adapts quickly to the change.

Below is the code snippet for the adversarial reward function and the corresponding experiment:

```
def adversarial_reward(t, arm, T, epsilon=0.01):
    if t < T/2:
        return 1 if arm == 1 else 0
    else:
        if arm == 0:
            return 1
        else:
            return 1 if np.random.rand() < epsilon else 0
```

```
def run_experiment_break(T=10000, n_runs=20, epsilon=0.01):
    all_regrets_ucb = np.zeros((n_runs, T))
    all_regrets_exp3 = np.zeros((n_runs, T))
    means = [0.5, 0.9]  # reference means, but actual rewards come from
   adversarial_reward

    for r in range(n_runs):
        reg_ucb = run_ucb(T, means, version=’modified’, nonstationary=
   True,
                              worst_case=True, epsilon=epsilon)
        all_regrets_ucb[r,:] = reg_ucb

        reg_e3 = run_exp3(T, means, nonstationary=True, worst_case=True,
   epsilon=epsilon)
        all_regrets_exp3[r,:] = reg_e3

    % (Plotting code is omitted here for brevity)
    return all_regrets_ucb, all_regrets_exp3
```

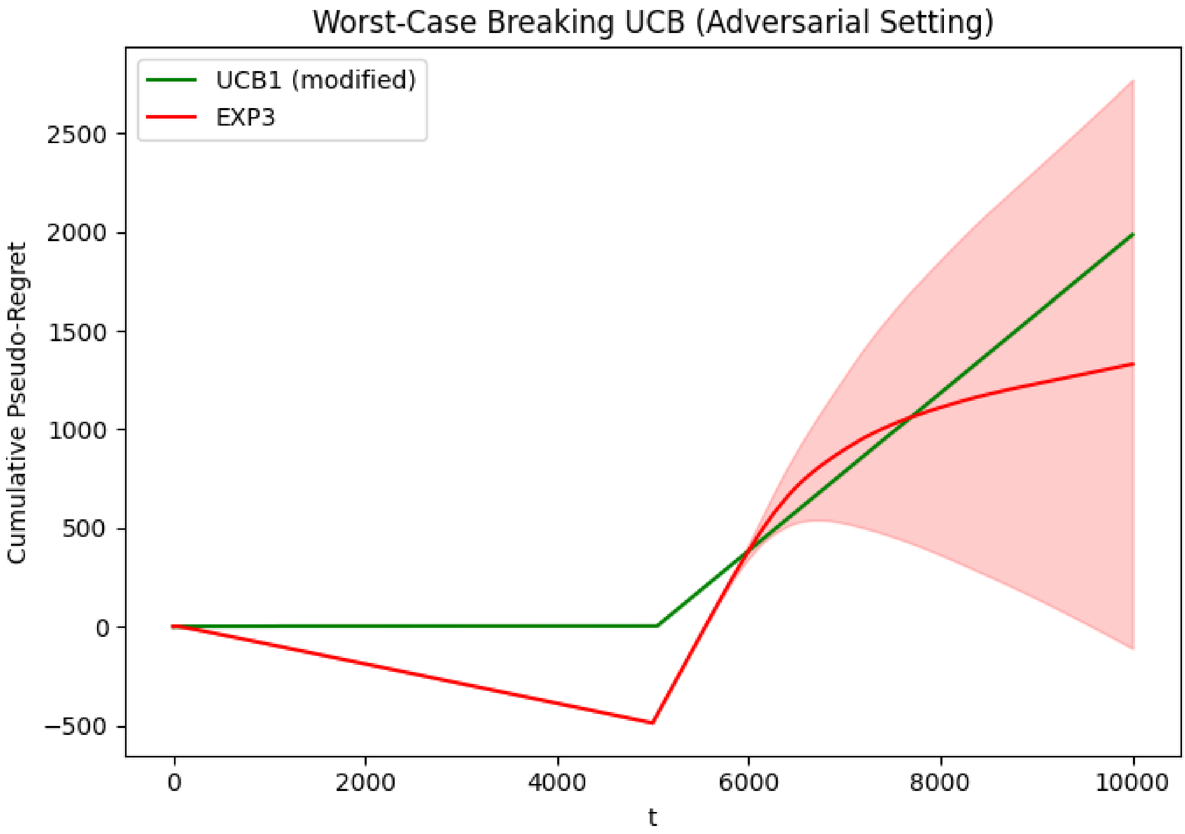

Figure 6: Worst-case adversarial scenario designed to break UCB1 (modified). Arm 1 is good for $t < T/2$, then flips, forcing UCB1 to suffer linear regret while EXP3 adapts faster.

Discussion:
In this adversarial setting:

- For $t < T/2$, arm 1 is heavily favored, so UCB1 builds a high empirical mean for arm 1.
- When $t \geq T/2$, the rewards flip, but UCB1 is slow to update its estimates and continues to select arm 1, which now almost always returns 0.
- EXP3, however, adjusts its probabilities based on the observed losses and quickly shifts its preference to arm 0.

It is noteworthy that breaking UCB is considerably harder than breaking Follow-The-Leader (FTL). In previous assignments, a simple repeating sequence of 0's and 1's was enough to break FTL via deterministic tie-breaking. However, UCB's exploration bonus and reliance on empirical means make it more robust, and for this reason a specifically crafted adversarial sequence (as shown above) is necessary to force it into linear regret.