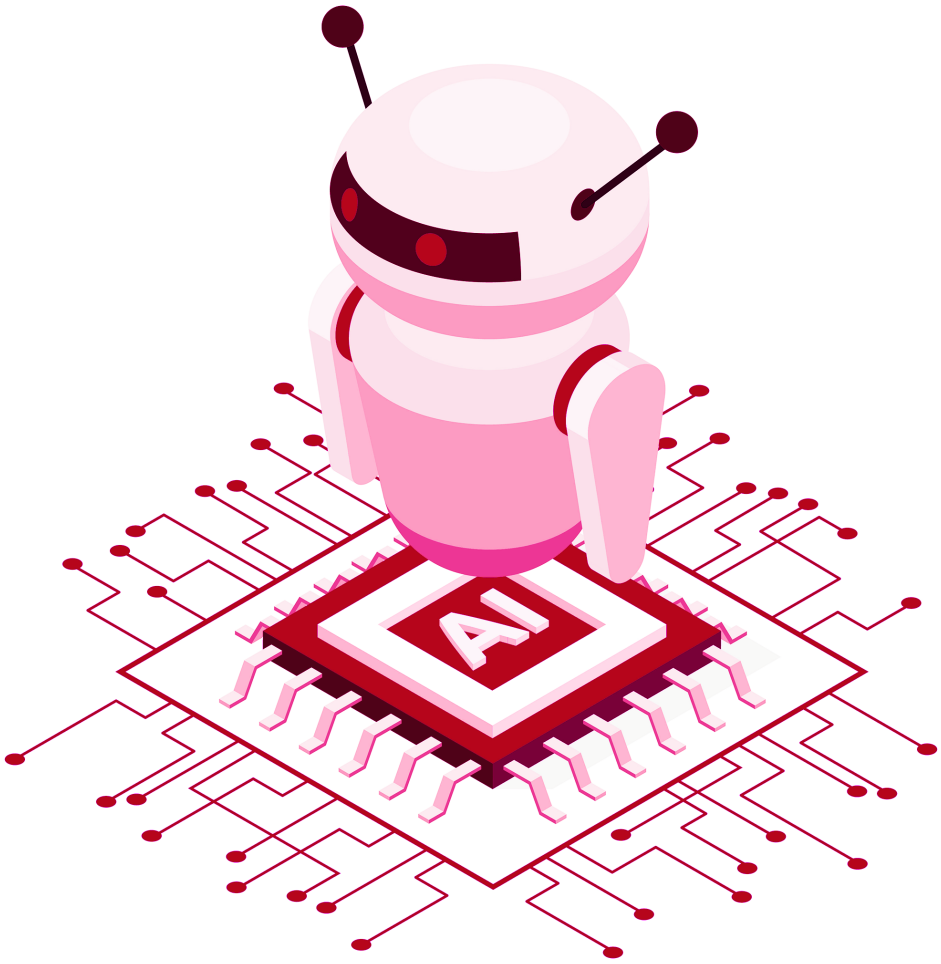

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 6:

Multi-Armed Bandits

By:

[Asemaneh Nafe]

[400105285]

RIML

Deep RL Course

Spring 2025

Contents

1 Task 1: Oracle Agent 1

2 Task 2: Random Agent (RndAg) 1

3 Task 3: Explore-First Agent (ExpFstAg) 1

4 Task 4: UCB Agent (UCB_Ag) 2

5 Task 5: Epsilon-Greedy Agent (EpsGdAg) 2

6 Task 6: LinUCB Agent (Contextual Bandits) 3

7 Task 7: Final Comparison and Analysis 3

8 Task 8: Final Deep-Dive Questions 4

Grading

The grading will be based on the following criteria, with a total of 105.25 points:

Task	Points
Task 1: Oracle Agent	3.5
Task 2: Random Agent	2
Task 3: Explore-First Agent	5.75
Task 4: UCB Agent	7
Task 5: Epsilon-Greedy Agent	2.25
Task 6: LinUCB Agent	27.5
Task 7: Final Comparison and Analysis	6
Task 8: Final Deep-Dive Questions	41.25
Clarity and Quality of Code	5
Clarity and Quality of Report	5
Bonus 1: Writing your report in Latex	10

1 Task 1: Oracle Agent

- The Oracle uses privileged information to determine the maximum expected reward.
- **TODO:** Compute the oracle reward (2 points).
- Questions:
  - What insight does the oracle reward provide? (0.75 points)
    The oracle reward tells us the absolute upper limit of performance that any agent could achieve in this environment. Since the Oracle always pulls the best possible arm, its expected per-step reward reflects the highest average reward achievable. This acts as a benchmark or gold standard against which we can compare the performance of learning agents. If another agent performs close to the oracle, it means it has learned the optimal policy effectively. On the other hand, a big gap shows that the agent still struggles with exploration or exploitation.
  - Why is the oracle considered "cheating"? (0.75 points) The oracle is considered "cheating" because it has access to information that a real-world agent wouldn't know in advance—namely, the exact success probabilities of each arm. In real applications, agents must learn about the environment through trial and error. The Oracle doesn't need to explore or make uncertain decisions; it skips the learning phase entirely. So, while it helps us understand what's theoretically possible, it's not realistic to use as a practical solution in most real scenarios.

2 Task 2: Random Agent (RndAg)

- This agent selects actions uniformly at random.
- **TODO:** Choose a random action (1 point).
- Questions:
  - Why is its reward lower and highly variable? (0.25 points)
    The random agent doesn't try to learn which arms are better—it just picks actions at random every time. This means it has no preference for high-reward arms and might frequently choose poor ones. Because of this, its average reward tends to be lower compared to agents that actively learn from feedback. Also, since the actions are chosen blindly, the results vary a lot across different runs. Sometimes it may hit a good arm more often by chance, and other times it might keep choosing weak ones, leading to high variability in performance.
  - How could the agent be improved without learning? (0.75 points)
    Even without actual learning, a random agent could be improved slightly by incorporating some simple rules or biases. For example, it could avoid repeating the same arm too often by cycling through arms more evenly (e.g., round-robin selection). Another idea is to assign fixed probabilities to each arm based on some prior belief or heuristic, instead of choosing uniformly at random. While these approaches don't adapt over time, they can prevent the agent from getting stuck in poor action patterns purely due to bad luck.

3 Task 3: Explore-First Agent (ExpFstAg)

- The agent explores randomly for a fixed number of steps and then exploits the best arm.
- **TODOs:**
  - Update Q-value (3 points)
  - Choose action based on exploration versus exploitation (1 point)
- Questions:

  - Why might a short exploration phase lead to fluctuations? (0.75 points)
    Because the agent is choosing actions completely at random during those first few steps, it has very limited information and no clear sense of which arms are good or bad. With only 5 random samples, it's quite possible that it doesn't pull the best arm at all—or only pulls it once by chance. As a result, when it switches to exploitation, it might latch onto an arm that seemed promising based on very limited data but isn't actually optimal. This can cause sudden jumps or drops in the reward curve early on, as the agent adjusts from exploration to what might be a poorly informed exploitation phase.
  - What are the trade-offs of using a fixed exploration phase? (1 point) A fixed exploration phase is simple and easy to implement, and it ensures that the agent gathers some initial data before committing to a choice. However, it's also rigid. If the exploration period is too short, the agent might not discover the best arm and get stuck exploiting a suboptimal one. If it's too long, the agent wastes valuable steps exploring arms that are clearly worse, lowering its average reward. This lack of adaptability is a key downside—there's no mechanism to adjust based on what the agent is learning in real-time. So, the performance heavily depends on tuning that one parameter maxex, which may not generalize well across different environments.

4 Task 4: UCB Agent (UCB_Ag)

- Uses an exploration bonus to balance learning.
- **TODOs:**
  - Update Q-value (3 points)
  - Compute the exploration bonus (4 points)

5 Task 5: Epsilon-Greedy Agent (EpsGdAg)

- Selects the best-known action with probability $1 - \varepsilon$ and a random action with probability ε.
- **TODO:** Choose a random action based on ε (1 point)
- Questions:
  - Why does a high ε result in lower immediate rewards? (0.5 points)
    A high ϵ value results in more exploration, meaning the agent frequently selects random actions rather than always exploiting the best-known option. While exploration allows the agent to discover potentially better arms (actions), it can also lead to suboptimal choices in the short term, as the agent is sometimes selecting actions that are not yielding the highest rewards. This randomness in action selection typically causes lower immediate rewards because the agent is not always leveraging the most rewarding arms it has learned about so far.
  - What benefits might decaying ε over time offer? (0.75 points)
    Decaying ϵ over time offers a balance between exploration and exploitation. Early in the process, the agent is unsure about the environment and benefits from exploring to discover which arms (actions) yield the best rewards. As the agent learns more about the environment, reducing ϵ allows it to shift focus to exploitation, where it increasingly favors the best-known actions. This decay helps avoid the inefficiency of constant exploration, and by gradually shifting to exploitation, the agent can maximize rewards based on accumulated knowledge. Additionally, decaying ϵ prevents the agent from getting stuck in exploration indefinitely and helps it converge toward the optimal solution over time.

6 Task 6: LinUCB Agent (Contextual Bandits)

- Leverages contextual features using a linear model.
- **TODOs:**
  - Compute UCB for an arm given context (7 points)
  - Update parameters A and b for an arm (7 points)
  - Compute UCB estimates for all arms (7 points)
  - Choose the arm with the highest UCB (4 points)
- Questions:
  - How does LinUCB leverage context to outperform classical methods? (1.25 points) LinUCB outperforms classical bandit algorithms by incorporating context (features) into the decision-making process. While traditional bandit algorithms like Epsilon-Greedy or UCB only consider past rewards, LinUCB uses the context to better estimate the expected reward for each arm. By computing an upper confidence bound (UCB) based on both past rewards and the context features, LinUCB can adaptively choose actions that are more informed, rather than relying solely on exploration or exploitation. This allows the agent to make smarter decisions, particularly in environments where the reward depends on complex, context-specific factors. In situations with rich, structured data, such as online recommendation systems or contextual advertising, LinUCB can make better decisions than simpler models, leading to improved performance.
  - What role does the α parameter play in the exploration bonus? (1.25 points)
    The α parameter in LinUCB controls the balance between exploration and exploitation by adjusting the exploration bonus. The exploration bonus is calculated as:

$$\alpha \cdot \sqrt{x_t^T A_a^{-1} x_t}$$

    where x_t is the context vector, and A_a^{-1} is the inverse of the matrix associated with arm a. The parameter α determines how much weight is given to this bonus. A higher α value increases the exploration component, meaning the agent will be more likely to explore less familiar arms, as the uncertainty in the reward estimation is given more weight. Conversely, a smaller α reduces the exploration bonus, making the agent more exploitative by focusing on actions with known, high expected rewards. Therefore, α directly influences how much the agent values trying new actions versus sticking to known successful ones. By tuning α, the agent can be adjusted for more aggressive exploration or more focused exploitation, depending on the task and environment.

7 Task 7: Final Comparison and Analysis

- **Comparison:** UCB vs. Explore-First agents.
  - Under what conditions might an explore-first strategy outperform UCB? (1.25 points) In very short time horizons, explore-first agents can outperform UCB because they focus their exploration early, then exploit consistently. If the best arm is correctly identified during the brief exploration phase, the remaining steps gain higher rewards with no further exploration cost.
    UCB, although asymptotically optimal, continues to explore based on uncertainty. This may lead to unnecessary exploration in the short term, especially when it is confident enough but still adds exploration bonuses.
    If the differences between arm reward probabilities (p_i) are large and easy to detect, even short exploration (like maxex $= 10$) might be enough for ExpFstAg to identify the best arm. UCB

might still spend time checking suboptimal arms to maintain theoretical guarantees.
infact An explore-first strategy might outperform UCB in certain practical situations where specific problem characteristics favor a simpler, more front-loaded approach to exploration. These conditions include:
Short Time Horizons: When the number of total steps is small, UCB's exploration bonus can lead to over-exploration of suboptimal arms early on. An explore-first agent, with a fixed and short exploratory phase, can quickly switch to exploitation and potentially gain higher short-term cumulative rewards.
Clear Gaps Between Arms: If one arm is significantly better than the others, even a brief exploration phase may be sufficient to identify it. In such scenarios, UCB might continue to allocate actions to clearly suboptimal arms due to its uncertainty estimates, while the explore-first agent can lock onto the best arm sooner.
Low Reward Variance: When reward noise is low and samples are reliable, fewer data points are needed to confidently estimate an arm's value. An explore-first agent benefits from this by needing fewer steps to identify the best option, whereas UCB may still "waste" steps on exploration due to its mathematical formulation.
Simplicity and Efficiency Constraints: UCB requires continuous updates to confidence bounds and logs, which can be computationally expensive. An explore-first strategy is more efficient and deterministic, making it more suitable for environments with strict computation or memory limitations.

  - How do design choices affect short-term vs. long-term performance? (1.25 points)
    Explore-First (ExpFstAg):
    Short-Term: Can perform well if maxex is appropriately tuned to identify the best arm quickly. But if exploration is too short, it risks locking into a suboptimal arm.
    Long-Term: Not adaptive. Once it enters exploitation, it never revisits other arms—even if initial estimates were wrong. Poor performance if initial samples are misleading.
    UCB:
    Short-Term: Might underperform due to its continual exploration—revisiting arms it should have ruled out. This adds regret early on.
    Long-Term: Performs better due to logarithmic regret growth. It adapts dynamically to new information and converges to optimal behavior.

- Impact of extending the exploration phase (e.g., 20 vs. 5 steps). (1.5 points total)
- Discussion on why ExpFstAg might sometimes outperform UCB in practice. (2 points)

8 Task 8: Final Deep-Dive Questions

- **Finite-Horizon Regret and Asymptotic Guarantees** (4 points)

  Many algorithms (e.g., UCB) are analyzed using asymptotic (long-term) regret bounds. In a finite-horizon scenario (say, 500–1000 steps), explain intuitively why an algorithm that is asymptotically optimal may still yield poor performance. What trade-offs arise between aggressive early exploration and cautious long-term learning? Deep Dive: Discuss how the exploration bonus, tuned for asymptotic behavior, might delay exploitation in finite time, leading to high early regret despite eventual convergence.
  In a finite-horizon setting, even an algorithm that's asymptotically optimal—like UCB—can struggle in the short term. The core issue lies in how these algorithms balance exploration and exploitation. They're designed with the long game in mind, meaning they explore thoroughly to guarantee that, eventually, they'll identify and stick with the best option. But in situations where time is limited,

this strategy can backfire.
Think of it like this: early on, algorithms like UCB tend to play it safe by exploring all the available arms, even those that don't seem particularly promising. This conservative approach ensures that no potentially good arm is overlooked. However, it comes at a cost—because the algorithm keeps exploring, it may delay focusing on the arm that already appears to be the best. As a result, in the short run, it racks up regret by not exploiting the higher-reward option as quickly as it could have.
This trade-off creates a tension between early exploration and timely exploitation. On one hand, aggressive exploration upfront is useful because it equips the algorithm with enough data to make informed decisions later on. In the long run, that reduces regret because it ensures the agent isn't misled by initial randomness or noise. But in a finite-horizon scenario, there may not be enough time to benefit from that careful learning. The agent could end up spending a significant portion of its limited time exploring mediocre options, missing the opportunity to capitalize on the better arm. Conversely, if the algorithm were to cut exploration short and start exploiting what it currently believes is best, it might perform better in the short term—but at the risk of never discovering the truly optimal choice. This is especially problematic if early reward estimates are misleading due to high variance or noise.
At the heart of this is the exploration bonus—the extra reward UCB gives to lesser-known arms. This bonus encourages trying out less-explored options, which is great for long-term learning. But in a finite setting, that same bonus might push the algorithm to waste time on arms that are clearly underperforming, simply because they haven't been tried enough yet. So while the algorithm is designed to perform well eventually, that eventual success might not arrive within the time frame you're working with—making it look inefficient or even naïve in the short term.
Ultimately, the problem is that the exploration bonus, while excellent for asymptotic behavior, isn't always tuned for immediate performance. And in practical scenarios where decisions need to be good quickly—think ad placements, clinical trials, or financial decisions—this delay can have real costs.

- **Hyperparameter Sensitivity and Exploration–Exploitation Balance** (4.5 points)

  Consider the impact of hyperparameters such as ϵ in ϵ-greedy, the exploration constant in UCB, and the α parameter in LinUCB. Explain intuitively how slight mismatches in these parameters can lead to either under-exploration (missing the best arm) or over-exploration (wasting pulls on suboptimal arms). How would you design a self-adaptive mechanism to balance this trade-off in practice? Deep Dive: Provide insight into the "fragility" of these parameters in finite runs and how a meta-algorithm might monitor performance indicators (e.g., variance in rewards) to adjust its exploration dynamically. In practice, the performance of bandit algorithms can hinge quite a

  bit on their hyperparameters. These parameters control how the agent balances exploration and exploitation, and small misconfigurations can lead to significantly different outcomes—especially when you're working within a limited time frame.
  Take ϵ in the ϵ-greedy algorithm. If ϵ is set too low, the agent will mostly exploit what it already knows and might never give lesser-known arms a fair chance. This can mean missing out on the optimal arm entirely, just because the algorithm didn't explore enough. On the flip side, if ϵ is too high, the agent spends too much time trying out different arms—even when it's fairly clear which one is best. That just wastes steps and hurts performance. Ideally, ϵ should start relatively high to encourage early exploration and then decay over time, letting the agent focus more on exploitation as it learns.
  The same kind of trade-off applies to the exploration constant in UCB. A low constant makes the algorithm greedy—it favors arms with high average rewards and doesn't explore much. That might

sound good in the short term, but if there's not enough exploration early on, the algorithm might never find the truly best option. A high constant, on the other hand, can make the agent overly cautious, spending too much time sampling uncertain (but unpromising) arms. The key is to find a balance that matches the task and the horizon.

Then there's α in LinUCB, which plays a similar role. When α is too small, the agent sticks to familiar arms too early. If it's too large, the agent wastes time chasing uncertainty that doesn't lead to better outcomes. Tuning α right means finding a sweet spot where the algorithm explores just enough to learn, but not so much that it loses time and reward.

Because of these challenges, one promising idea is to design a self-adaptive mechanism—something that can tune these hyperparameters on the fly, based on how the agent is doing. For instance, the algorithm could monitor reward variance across arms. If the variance is high, it might suggest the agent is still uncertain and should explore more. If it's low, the agent could safely exploit. Similarly, if the regret is growing rapidly, that might be a sign that exploration needs to increase.

This kind of adjustment could be handled by a meta-algorithm that tracks these signals—like how often each arm is selected, how rewards fluctuate, or how regret changes over time—and adjustsϵ, the UCB constant, or α accordingly. When the agent starts out and the environment is still unknown, the meta-algorithm can promote exploration. As the agent learns more, the meta-algorithm can dial back the exploration, letting the agent focus on what works.

This is especially important in finite-horizon settings, like when you only have 500 or 1000 rounds. You don't have infinite time to figure things out. A hyperparameter that's slightly off can really mess things up. If exploration is too timid, the agent might never find the best arm. If it's too aggressive, it might waste most of its time on bad choices. A self-adaptive approach helps prevent both of these extremes by tuning behavior as the agent learns.

- **Context Incorporation and Overfitting in LinUCB** (4 points)

  LinUCB uses context features to estimate arm rewards, assuming a linear relation. Intuitively, why might this linear assumption hurt performance when the true relationship is complex or when the context is high-dimensional and noisy? Under what conditions can adding context lead to worse performance than classical (context-free) UCB? Deep Dive: Discuss the risk of overfitting to noisy or irrelevant features, the curse of dimensionality, and possible mitigation strategies (e.g., dimensionality reduction or regularization).

- **Adaptive Strategy Selection** (4.25 points)

  Imagine designing a hybrid bandit agent that can switch between an explore-first strategy and UCB based on observed performance. What signals (e.g., variance of reward estimates, stabilization of Q-values, or sudden drops in reward) might indicate that a switch is warranted? Provide an intuitive justification for how and why such a meta-strategy might outperform either strategy alone in a finite-time setting. Deep Dive: Explain the challenges in detecting when exploration is "enough" and how early exploitation might capture transient improvements even if the long-term guarantee favors UCB.

- **Non-Stationarity and Forgetting Mechanisms** (4 points)

  In non-stationary environments where reward probabilities drift or change abruptly, standard bandit algorithms struggle because they assume stationarity. Intuitively, explain how and why a "forgetting" or discounting mechanism might improve performance. What challenges arise in choosing the right decay rate, and how might it interact with the exploration bonus? Deep Dive: Describe the delicate balance between retaining useful historical information and quickly adapting to new trends, and the potential for "chasing noise" if the decay is too aggressive.

- **Exploration Bonus Calibration in UCB** (3.75 points)

  The UCB algorithm adds a bonus term that decreases with the number of times an arm is pulled. Intuitively, why might a "conservative" (i.e., high) bonus slow down learning—even if it guarantees asymptotic optimality? Under what circumstances might a less conservative bonus be beneficial, and what risks does it carry? Deep Dive: Analyze how a high bonus may force the algorithm to continue sampling even when an arm's estimated reward is clearly suboptimal, thereby delaying convergence. Conversely, discuss the risk of prematurely discarding an arm if the bonus is too low.

- **Exploration Phase Duration in Explore-First Strategies** (4 points)

  In the Explore-First agent (ExpFstAg), how does the choice of a fixed exploration period (e.g., 5 vs. 20 steps) affect the regret and performance variability? Provide a scenario in which a short exploration phase might yield unexpectedly high regret, and another scenario where a longer phase might delay exploitation unnecessarily. Deep Dive: Discuss how the "optimal" exploration duration can depend heavily on the underlying reward distribution's variance and the gap between the best and other arms, and why a one-size-fits-all approach may not work in practice.

- **Bayesian vs. Frequentist Approaches in MAB** (4 points)

  Compare the intuition behind Bayesian approaches (such as Thompson Sampling) to frequentist methods (like UCB) in handling uncertainty. Under what conditions might the Bayesian approach yield superior practical performance, and how do the underlying assumptions about prior knowledge

influence the exploration–exploitation balance? Deep Dive: Explore the benefits of incorporating prior beliefs and the risk of bias if the prior is mis-specified, as well as how Bayesian updating naturally adjusts the exploration bonus as more data is collected.

- **Impact of Skewed Reward Distributions** (3.75 points)

  In environments where one arm is significantly better (skewed probabilities), explain intuitively why agents like UCB or ExpFstAg might still struggle to consistently identify and exploit that arm. What role does variance play in these algorithms, and how might the skew exacerbate errors in reward estimation? Deep Dive: Discuss how the variability of rare but high rewards can mislead the agent's estimates and cause prolonged exploration of suboptimal arms.

- **Designing for High-Dimensional, Sparse Contexts** (5 points)

  In contextual bandits where the context is high-dimensional but only a few features are informative, what are the intuitive challenges that arise in using a linear model like LinUCB? How might techniques such as feature selection, regularization, or non-linear function approximation help, and what are the trade-offs involved? Deep Dive: Provide insights into the risks of overfitting versus underfitting, the increased variance in estimates from high-dimensional spaces, and the potential computational costs versus performance gains when moving from a simple linear model to a more complex one.