# COMP0037 2024 / 2025 Robotic Systems

# Lab 01: Multi-Arm Bandits

COMP0037 Teaching Team

January 17, 2025

## Overview

In this lab, we will explore the $k$-arm bandit problem. The bandit problem is one of the most basic problems encountered in policy learning. Despite its simplicity, it illustrates many extremely important practical problems. In the lab you will implement a number of algorithms for agents to solve the bandit problem. The lab also presents an opportunity for you to become familiar with the GUI environment and (slightly) with OpenAI Gym.

The lab itself consists of a lot of small tasks. Each one shouldn't take much time to complete, but should give you a chance to try coding, exploring the API and some of the characteristics. You might notice that some files contain solutions to earlier tasks; since this is an unmarked lab this is a deliberate feature and will allow you to check your earlier work as you progress.

Note that sometimes that part a can occur later in the file than part b.

The setup consists of the following:

1. There are a set of bandits, each of which corresponds to a slot machine.
2. The reward from the $i$th bandit is drawn from a Gaussian distribution with mean $\mu_i$ and covariance $\sigma_i^2$.
3. An environment is a collection of multiple bandits.
4. The action the agent can undertake is to select one of the bandits.
5. When bandit $i$ is selected, the reward is drawn from that bandit.

6. The agent is responsible for selecting arms, computing reward signals, and working out the policy it will use.

To carry out the lab, you will download and run some Python code and modify it to implement you answers. The code provided runs, but does not do anything interesting. Comments are provided in the code to suggest where changes should be made. These comments are based on the changes we made to get the assigned tasks work, but you might develop a solution which works differently.

# Preliminaries

The activities in this section focus on getting familiar with the basic system.

1. The file `test_bandit.py` contains code which will be used to test the properties of an individual bandit running on its own. In this part, you only need to modify this file.

   a. Modify the code to create an instance of a `Bandit` object with mean 1.0 and standard deviation 2.0.

   b. Change the number of times the arm of the bandit is pulled and explore how the mean and covariance change. What do notice about the rate of each? You might want to increase by a factor of 10 each time. If you want, you can instrument the code to measure the amount of time required as well.

   c. The code computes the mean reward using a "batch" in which we store all the reward signals and compute the mean at the end. An alternative approach is to use a recursive equation of the form:

   $$Q_{t+1}(a) = Q_t(a) + \frac{1}{t+1}\left(R_t(a) - Q_t(a)\right).$$

   Implement this form and compare it with your batch calculation.

   *Hint:* Both the batch and resursive forms should produce near-identical results (they shoudl agree to at least 8 decimal places) for both small and large numbers of samples. If the two do not produce the same value, check your use of indexing.

2. The file `test_environment.py` contains code which will be used to test the properties of multiple bandits in the environment. In this part, you only need to modify this file.

   a. With reference to the API for the `BanditEnvironment` class, create a bandit environment with four bandits. These bandits should have the following means and covariances:

| Bandit | Mean | Standard deviation |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 2 | 1 |
| 4 | 2 | 2 |

   b. Modify the method `run_bandits` so that it will iterate through each bandit in the environment and compute the mean and covariance of the rewards. Confirm these values are the same as the ones specified in the agent.

   *Hint:* Your code should *only* use the input variables specified. In particular, your solution must *not* be hardwired to the number of bandits in the environment. Rather, it must query the environment object and determine them directly.

3. The final part is to implement an agent. You will modify the files `random_action_agent.py` and `test_agent.py`

   a. Modify the implementation of `random_action_agent.py` so that it will pick a valid action at random.

      *Hint:* Because we are using OpenAI gym to describe our environment, the environment itself can be used to draw random actions. See https://gym.openai.com/docs/ for details.

   b. Modify `test_agent.py` to add plots to show the sequence of actions taken and the reward signals.

      *Hint:* Check `test_bandit.py` to see how to create a plot and draw values in it. To open multiple figures — either in the same or different windows — check the matplotlib documentation.

## Performance Measures

In this question, you will implement two common methods for assessing the performance of bandit agents. You will need to modify `bandits/performance_measures.py` and `evaluate_try_them_all.py`.

4. a. Modify `evaluate_try_them_all.py` to plot the percentage of correct actions. Experiment with different numbers of initial trials and see how this influences the number of correct actions. What do you think the curves are showing you?

   b. Implement the method `compute_regret` to compute the regret. Your method should only use the `environment` and the `reward_history` and should not be hard coded.

   *Hint:* Check `compute_percentage_of_optimal_actions_selected` and the public API of `BanditEnvironment`.

   c. Modify `evaluate_try_them_all.py` to compute and plot the regret. Experiment with different numbers of initial trials and see how this impacts the regret.

## $\epsilon$-greedy Search

5. The simplest method for performing random exploration is to use the $\epsilon$-Greedy algorithm. This is described in Slides 56–67 of Lecture 02.

    a. Modify the method `_compute_action` of `EpsilonGreedyAgent` to implement the $\epsilon$-greedy exploration algorithm.

    *Hint:* Recall that you either need to pick a random agent or the agent with the current best average action.

    b. Experiment with different values of $\epsilon$. What do you notice about the behaviour of the regret and percentage optimal selection algorithms?

    c. One common way to improve $\epsilon$-greedy is to damp or change exploration over time.

    Experiment with the agent implemented in `DampedEpsilonGreedyAgent`. What do you notice?

## Upper Confidence Bound

The UCB is another algorithm which attempts to find the optimal solution by taking account of the uncertainty. It is described in the lectures slides 73–84.

6. a. Modify `upper_confidence_bound_agent.py` to implement the UCB algorithm.

    b. Explore with different values of $c$ to see the impact on the rewards and the percentage of optimal actions taken.