Université Paris-Saclay
Institut Polytechnique de Paris
M2 MN

Year 2024-2025

TD-TP LC4MM – 1

Integrated Lab session
Learning and Control for Multimedia
—
Lab 1 – *Multi-armed bandit*

Warning : This practical course must be prepared by all students before the start of the session. The preparation includes reading the whole topic and understanding it. You should also think about the manipulation part and plan the tests you will perform to illustrate the lab.

This practical course requires a working Python environment, such as

— WinPython (recommanded), see `https://winpython.github.io`
— Anaconda (recommanded), see `https://www.anaconda.com`
— Pyzo, see `https://pyzo.org`
— Google Collaborate, see `https://colab.research.google.com`

1 Introduction

Multi-armed bandit problems are the simplest class of reinforcement learning problems to solve. This type of problem involves an agent who has a choice between several possible actions. Each action is associated with a random reward described by a probability distribution that is generally unknown. The agent's objective is to determine the action that will produce the maximum reward. For this purpose, several runs will be considered during each of which the agent can choose only one action at a time. The probability distribution characterising the reward associated with an action will be assumed to be time invariant.

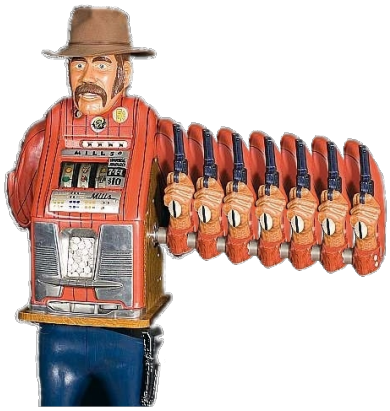

FIGURE 1 – Multi-armed Bandit

This type of problem owes its name of *multi-armed bandit* to the fact that it can be represented by k single-armed slot machines, each characterized by an unknown probability of winning. The task is to determine which machine produces the highest average payoff. Alternatively, we can represent a single slot machine with k arms, each characterized by an unknown probability of winning. Given, for example, 1000 one Euro coins, we need to determine strategy that maximises our gains (or minimises our losses).

One way to approach this problem is to select each slot machine in turn, keep track of the reward received, and then replay the machine that generated the highest reward. However, the reward offered by a machine is described by a probability distribution, which means that it may take several tries to find the best one. Each trial of a machine to find the best one takes us further away from maximising the total reward. This leads to a balance to be determined between exploration and exploitation.

The practical situations that can be modelled by a *multi-armed bandit* problem are varied. For example, there are problems of routing in communication networks, of selecting a treatment in medicine, of choosing the advertisement to be displayed when viewing a page on the Internet, and of controlling systems which are difficult to model.

2 Notations and problem formulation

We will consider k one-armed slot machines. At time n, the agent must choose an action $a \in \mathcal{A} = \{1, \ldots, k\}$, corresponding to the index of the machine whose arm he will pull. The objective is to choose the action which maximizes

$$q_n^*(a) = \mathbb{E}(R_n \mid A_n = a), \tag{1}$$

where $q_n^*(a)$ is the expected value of R_n knowing that the action a has been chosen at time n.

We will assume in the rest of the study that the rewards associated with an action a chosen several times can be described as realizations of independent and identically distributed random variables.

Preparation 1 *Assume that at time n, the action a has been chosen N_a times and that the rewards $R_1, \ldots, R_{N_a}$ have been obtained.*

1. *Which estimate $Q_n(a)$ of $q^*(a)$ can we get from $R_1, \ldots, R_{N_a}$?*
2. *Assuming that an estimate $Q_n(a)$ of $q^*(a)$ has been obtained from $R_1, \ldots, R_{N_a}$ and that a new reward R_{N_a+1} has been obtained after a has been chosen at time $n+1$, express $Q_{n+1}(a)$ as a function of $Q_n(a)$, of R_{N_a+1}, and of N_a.*
3. *Express $Q_{n+1}(a')$ using $Q_n(a')$ for an action a' that has not been chosen at time $n+1$.*

3 ε-greedy policy

An ε-*greedy* policy aims at time $n+1$ to select
— with a probability ε, an action uniformly at random
— with a probability $1-\varepsilon$, the action minimizing $Q_n(a)$

$$a_{n+1}^* = \arg\max_{a \in \mathcal{A}} Q_n(a) \tag{2}$$

and to update the estimate of $q^*(a)$ for all $a \in \mathcal{A}$.

When $\varepsilon = 0$, The action maximizing $Q_n(a)$ is always chosen. This is the greedy policy.

The python class `eps_bandit` described in Appendix A is a possible implementation of a *multi-arm bandit* with k arms for which an ε-greedy policy may be employed for the selection of the arm to pull.

The class constructor

def __init__(self, k, eps, runs, mu='random'):

defines the number k of arms of the multiarm bandit, the value of $varepsilon$, the number `runs` of runs within each realization, as well as the average value `mu` of the rewards associated with each arm. Several initializations are possible for `mu`. By default, the mean values of the rewards are realisations of Gaussian random variables of zero mean and unity variance.

The method

def pull(self):

pulls an arm according to an ε-greedy policy.

The method

def run(self):

performs `runs` arm selections using an ε-greedy policy.

The method

def reset(self):

allows different variables of the multiarm bandit to be reset without changing its number of arms or the average value of the rewards associated with each arm.

To use that class, make sure that the file `eps_bandit.py` is in your current folder. Then using Spyder or a Jupyter notebook, you have to import different modules.

```
# import modules
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from eps_bandit import *

%matplotlib inline
```

The `%matplotlib inline` is only useful with Jupyter or Google Colab.

We consider a *multi-arm bandit* with $k = 10$ arms. Each bandit realization consists of 1000 runs. We start with 10 realizations and an ε-greedy policy with $\varepsilon = 0.1$.

```
# Main part of the program
# Number of arms
k = 10
# Number of runs for each bandit realization
runs = 1000

# Array to store the average rewards as a function of the realization
eps_1_rewards = np.zeros(runs)

# Number of realizations of the bandit for the simulation
realizations = 10
```

For each realization, a new instance of `eps_bandit` is generated, then `runs` successive action choices are performed and the empirical mean of the rewards at each run are updated at the end of each realization.

```
# Run experiments
for i in range(realizations):
    # Initialize bandits
    eps_1 = eps_bandit(k, 0.1, runs)

    # Run experiments
    eps_1.run()

    # Update long-term averages among realizations
    eps_1_rewards = eps_1_rewards + (
        eps_1.reward - eps_1_rewards) / (i + 1)
```

The results are finally displayed.

```
plt.figure(figsize=(14,8))
plt.plot(eps_1_rewards, label="$\epsilon=0.1$")
plt.legend(bbox_to_anchor=(0.6, 0.5))
plt.xlabel("Run")
plt.ylabel("Average␣Reward")
```

```
plt.title("Average␣$\epsilon−greedy$␣Rewards␣after␣" + str(realizations)
    + "␣realizations")
plt.show()
```

Manipulation 1

1. *Implement the main program evaluating the average reward as a function of the number of realizations for an ε-greedy policy.*
2. *Compare the results obtained with* 10, 100, *and* 1000 *realizations.*
3. *Complete the main program by considering two* multi-arm bandits *with similar characteristics to those of the first* multi-arm bandit, *but operated for one with a greedy policy and for the other with a ε-greedy policy with $\varepsilon = 0.01$. To do this, we could for example add the commands*

   ```
   eps_0 = eps_bandit(k, 0.0, runs, eps_1.mu.copy())
   eps_01 = eps_bandit(k, 0.01, runs, eps_1.mu.copy())
   ```

   once the first multi-arm bandit *has been defined.*

   (a) *What is the point of having average values of the rewards associated with each arm equal to that of the initial multi-arm bandit ?*

   (b) *Compare the performance of the three considered policies.*

Manipulation 2 *We will try to assess the proportion of optimal actions chosen according to the policy.*

1. *Modify the previous program by adding a vector to store the average proportion of choices for each action, initialized as follows*

   ```
   # Array to store the average action selection among realizations
   eps_1_selection = np.zeros(k)
   ```

2. *Modify the program by generating a first* multi-arm bandit *with average reward values increasing from* 1 *to* k *using the command*

   ```
   eps_1 = eps_bandit(k, 0, runs, mu='sequence')
   ```

3. *At the end of each realization, update the vector storing the average number of choices for each action*

   ```
   # Average actions per realization
   eps_1_selection = eps_1_selection + (
       eps_1.k_n − eps_1_selection) / (i + 1)
   ```

   Do not forget to make these modifications for the other two policies.

4. *Compare the performance of the three policies considered with these new reward averages.*

5. *Visualise the proportion of choices for each of the actions using the following command*

   ```
   bins = np.linspace(0, k−1, k)

   plt.figure(figsize=(12,8))
   plt.bar(bins, eps_0_selection,
           width = 0.33, color='b', label="$\epsilon=0$")
   plt.bar(bins+0.33, eps_01_selection,
           width=0.33, color='g', label="$\epsilon=0.01$")
   plt.bar(bins+0.66, eps_1_selection,
           width=0.33, color='r', label="$\epsilon=0.1$")
   plt.legend(bbox_to_anchor=(1.2, 0.5))
   plt.xlim([0,k])
   plt.title("Actions␣Selected␣by␣Each␣Algorithm")
   plt.xlabel("Action")
   plt.ylabel("Number␣of␣Actions␣Taken")
   plt.show()
   ```

   to obtain an histogram and

   ```
   opt_per = np.array([eps_0_selection, eps_01_selection,
                       eps_1_selection]) / runs ∗ 100
   df = pd.DataFrame(opt_per, index=['$\epsilon=0$',
       '$\epsilon=0.01$', '$\epsilon=0.1$'],
                     columns=["a␣=␣" + str(x) for x in range(0, k)])
   print("Percentage␣of␣actions␣selected:")
   df
   ```

   to get a table.

6. *Comment the obtained results.*

4 ε-greedy policy with decreasing ε

The ε-greedy policies have an obvious flaw : they continue to explore for a ε proportion of the runs, regardless of the level of progress in the runs for a bandit realization. It would be better if they gradually settled on an optimal solution and continued to exploit it. To do this, we can introduce the ε-greedy policy with a decay of ε, which reduces the probability of exploration at each run. To do this, it is sufficient to define ε as a function of the run n for a given realization of the multi-arm bandit

$$\varepsilon_n = \frac{1}{1+\beta n} \tag{3}$$

where $\beta \in]0,1[$ is a parameter that determines the decreasing speed.

Manipulation 3

1. *Plot the evolution of ε_n as a function of n for different values of β.*
2. *Create a class* `eps_decay_bandit` *inspired by* `eps_bandit` *allowing to implement a* multi-arm bandit *operated with an ε-greedy with a decay of ε.*
3. *Compare the performances of the ε-greedy policy without and with decay of ε for various values of β.*

5 Greedy policy with optimistic initialization

An alternative to the ε-greedy policy is to encourage exploration by optimistically initializing the reward estimates and then applying a greedy policy at each run. This policy requires some knowledge of the possible values of the rewards.

It is not necessary to implement a new class. It is enough to use the class `eps_bandit` and to create an instance of *multi-arm bandit* operated with a greedy policy.

Manipulation 4

1. *Initialise a multi-arm bandit with a greedy policy by initialising the estimated rewards associated with each arm optimistically, for example using*

   ```
   # Initialize bandits
       oiv_bandit = eps_bandit(k, 0, runs)
       oiv_bandit.k_reward = np.repeat(5., k)
       oiv_bandit.k_n = np.ones(k)
   ```

2. *Compare the performance obtained with this policy and that obtained with the previous policies.*

A `eps_bandit` class

```
import numpy as np

class eps_bandit:
    '''
    epsilon-greedy k-bandit class

    Inputs
    =====================================================
    k: number of arms (int)
    eps: probability of random action 0 < eps < 1 (float)
```

```
    runs: number of runs (int)
    mu: set the average rewards for each of the k-arms.
        Set to "random" for the rewards to be selected from
        a normal distribution with mean = 0.
        Set to "sequence" for the means to be ordered from
        0 to k-1.
        Pass a list or array of length = k for user-defined
        values.
    '''

    def __init__(self, k, eps, runs, mu='random'):
        # Initialization of one instance of the class
        # Number of arms
        self.k = k
        # Search probability
        self.eps = eps
        # Number of runs
        self.runs = runs
        # Step count
        self.n = 0
        # Step count for each arm
        self.k_n = np.zeros(k)
        # Total mean reward
        self.mean_reward = 0
        self.reward = np.zeros(runs)
        # Mean reward for each arm
        self.k_reward = np.zeros(k)

        if type(mu) == list or type(mu).__module__ == np.__name__:
            # User-defined averages
            self.mu = np.array(mu)
        elif mu == 'random':
            # Draw means from probability distribution
            self.mu = np.random.normal(0, 1, k)
        elif mu == 'sequence':
            # Increase the mean for each arm by one
            self.mu = np.linspace(0, k-1, k)

    def pull(self):
        # Pull an arm (epsilon-greedy policy)
        # Generate random number
        p = np.random.rand()
        if self.eps == 0 and self.n == 0:
            a = np.random.choice(self.k)
        elif p < self.eps:
            # Randomly select an action
            a = np.random.choice(self.k)
        else:
            # Take greedy action
            a = np.argmax(self.k_reward)

        # Random generation of reward
```

```
        reward = np.random.normal(self.mu[a], 1)

        # Update counts
        self.n += 1
        self.k_n[a] += 1

        # Update total
        self.mean_reward = self.mean_reward + (
            reward − self.mean_reward) / self.n

        # Update results for a_k
        self.k_reward[a] = self.k_reward[a] + (
            reward − self.k_reward[a]) / self.k_n[a]

def run(self):
    # Run a simulation for one bandit realization
    for i in range(self.runs):
        self.pull()
        self.reward[i] = self.mean_reward

def reset(self):
    # Resets results while keeping settings
    self.n = 0
    self.k_n = np.zeros(self.k)
    self.mean_reward = 0
    self.reward = np.zeros(self.runs)
    self.k_reward = np.zeros(self.k)
```