ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

# Python et IA

Apprentissage par renforcement : bandit manchot à $K$ bras

A. Lebret, P. Lefebvre

2025-2026

**Objectif**

Ce TP introduit l'apprentissage par renforcement à travers une variante du problème du bandit manchot à $K$ bras.
Vous y découvrirez :

- la notion de stratégie (règle de décision) ;
- le rôle des récompenses et de l'expérience accumulée ;
- le compromis entre l'exploration et l'exploitation ;
- une stratégie classique : la stratégie $\epsilon$-glouton.

L'objectif n'est pas de couvrir toutes les méthodes d'apprentissage par renforcement, mais de comprendre sur un exemple concret comment un agent apprend à agir dans un environnement inconnu, comment on mesure ses performances et pourquoi il doit parfois explorer au lieu de toujours exploiter la meilleure action connue.

Durée estimée : 2 heures sans bonus

# 1. Introduction

L'apprentissage supervisé (régression, classification) repose sur des exemples d'entrées et de sorties pour lesquels on dispose d'un jeu de données annoté, et à l'aide desquels on cherche une fonction approchant au mieux la relation.

L'apprentissage par renforcement, au contraire, repose sur l'idée d'un agent qui interagit avec un environnement sans le connaître *a priori*. L'agent qui agit choisit une action à effectuer, puis reçoit une récompense (ou pas), et en fonction de cette récompense met à jour sa stratégie.

Dans ce TP, nous étudions une variante du bandit manchot à plusieurs bras dans laquelle les probabilités de gain pour chacun des bras changent avec le temps. Un agent efficace doit alors s'adapter continuellement afin de trouver quel bras est susceptible à un temps $t$ de lui fournir la meilleure récompense.

Cela introduit naturellement deux questions :

1. Comment estimer correctement la valeur d'une action quand l'environnement change ?
2. Comment équilibrer les étapes d'exploration de l'environnement, puis son exploitation ?

# 2. Environnement : le bandit à $K$ bras

Nous modélisons un bandit à $K$ bras par une classe `KArmedBandit` fournie dans le fichier « `bandit_env.py` ».

```
# bandit_env.py
import numpy as np
```

```python
class KArmedBandit:
    """
    Bandit à K bras, chaque bras retournant une récompense entre 0 et 1
    qui évolue dans le temps.
    """
    def __init__(self, K: int, rng=None):
        self.K = K
        self.rng = np.random.default_rng() if rng is None else rng

        # Phase initiale
        phases = self.rng.uniform(0, 2*np.pi, size=K)
        self.phases = phases

    def true_probs(self, t: int):
        """
        Probabilités réelles à l'instant t.
        Variation sinusoïdale lente.
        """
        return 0.5 + 0.4 * np.sin(0.005 * t + self.phases)

    def step(self, action: int, t: int) -> float:
        """
        Joue le bras 'action' à l'instant t.
        Retourne une récompense binaire (0 ou 1) selon une loi de Bernoulli
        de paramètre p(t) = probabilité vraie du bras à l'instant t.
        """
        p = self.true_probs(t)[action]
        return float(self.rng.random() < p)
```

Chaque bras oscille entre 0.1 et 0.9 et aucun bras n'est optimal en permanence. Le bras retournant la meilleure récompense change progressivement et l'agent doit donc réagir rapidement aux variations.

## 3. Travail préparatoire

1. Décompressez l'archive « `tp9_bandit.zip` » que vous récupérerez sur la page de cours. Vous devriez obtenir l'arborescence suivante :

```
tp9_bandit/
|-- src/
|   |-- bandit_env.py        # Classe du bandit manchot
|   |-- agent.py             # Agent à compléter
|   |-- main_bandit.py       # Script principal à compléter
|   |-- plot_utils.py        # Fonctions utilitaires pour les tracés de courbes
|   |-- __init__.py
|-- requirements.txt
```

Créez un environnement virtuel pour cette séance :

```
python3 -m venv env_tp9
source env_tp9/bin/activate
pip install -r requirements.txt
```

## 4. Exercices

### Exercice n°1 : Interaction avec le bandit manchot - stratégie aléatoire

Dans un premier temps, vous allez interagir avec le bandit à l'aide d'une stratégie aléatoire : à chaque pas, l'agent choisit un bras au hasard, sans exploiter ce qu'il a appris auparavant. La stratégie aléatoire revient à estimer les récompenses par une méthode de type Monte Carlo, c'est-à-dire par moyenne d'échantillons indépendants.

1. Compléter le fichier « `src/main_bandit.py` » qui :
   — importe la classe `KArmedBandit` depuis « `bandit_env.py` » ;
   — instancie un bandit à 4 bras de la manière suivante :

   ```
   from bandit_env import KArmedBandit

   env = KArmedBandit(K=4)
   ```

2. Simulez $T = 2000$ pas de temps avec une stratégie aléatoire :
   — à chaque pas, choisissez une action `a_t` uniforme dans 0, 1, 2, 3 (le numéro du bras) ;
   — appelez `env.step(a_t, t)` pour obtenir la récompense correspondant à l'action `a_t` au temps `t` ;
   — stockez les récompenses dans une liste ou un tableau.
3. Calculez et affichez :
   — la récompense moyenne obtenue sur les $T$ pas ;
   — le nombre de fois où chaque bras a été joué.
4. Tracez (avec *Matplotlib*) la récompense moyenne cumulée en fonction du temps.

> Pour calculer la récompense moyenne cumulée jusqu'au temps $t$, utilisez la fonction `cumsum()` de *Numpy* :
>
> ```
> rewards = np.array(rewards)  # taille T
> cumulative_mean = np.cumsum(rewards) / np.arange(1, T+1)
> ```
>
> Puis tracez `cumulative_mean` en fonction de `np.arange(T)`.

**Questions** :

— La stratégie aléatoire finit-elle par privilégier le meilleur bras ?
— Pourquoi cette stratégie ne tire-t-elle pas parti de l'évolution des probabilités ?

## Exercice n°2 : Agent $\epsilon$-glouton

Pour chaque bras $k \in \{0, 1, \dots, K-1\}$, on note $Q_k$ l'estimation de la récompense moyenne de ce bras. On cherche à estimer la vraie probabilité de gain $p_k(t)$ qui varie avec le temps.

Un agent $\epsilon$-glouton utilise ces estimations pour guider ses choix selon la stratégie suivante :

- avec probabilité $\epsilon$ : exploration (choisir un bras au hasard) ;
- avec probabilité $1-\epsilon$ : exploitation (choisir le bras $k$ tel que $Q_k$ est maximal).

#### 0.0.1 Mise à jour de l'estimation

Contrairement à un environnement stationnaire où l'on utiliserait la moyenne arithmétique classique :

$$Q_k = \frac{1}{N_k} \sum_{i=1}^{N_k} r_i^{(k)}$$

(où $N_k$ est le nombre de fois où le bras $k$ a été joué), cette approche ne convient pas ici car les probabilités $p_k(t)$ changent avec le temps. Les anciennes observations deviennent obsolètes.

On utilise donc une moyenne mobile exponentielle qui accorde plus de poids aux récompenses récentes :

$$Q_k \leftarrow Q_k + \alpha\,(r - Q_k)$$

où :

- $r$ est la récompense observée (0 ou 1) après avoir joué le bras $k$ ;
- $\alpha \in ]0, 1]$ est le **taux d'apprentissage** qui contrôle la vitesse d'adaptation de notre agent.

Cette règle peut être réécrite comme $Q_k \leftarrow (1-\alpha)Q_k + \alpha\,r$, montrant que la nouvelle estimation est une combinaison pondérée de l'ancienne valeur et de la nouvelle observation.

> **Lien avec l'apprentissage par renforcement général**
> La notation $Q$ vient de la *Q-function* (ou fonction de valeur d'action) en apprentissage par renforcement, notée $Q(s, a)$ où $s$ est un état et $a$ une action. Dans le problème du bandit, il n'y a qu'un seul état (implicite), donc $Q(s, a)$ se réduit à $Q_a$ ou $Q_k$ : la valeur associée à chaque action/bras.

### 2.1. Implémentation de l'agent

Complétez le fichier « `src/agent.py` » :

```
# agent.py
import numpy as np

class EpsilonGreedyAgent:
```

```python
    """
    Agent epsilon-glouton pour un bandit à K bras.
    Il maintient une estimation de la récompense moyenne Q_k pour chaque bras k.
    """
    def __init__(self, K: int, epsilon: float, alpha: float, rng=None):
        """
        K       : nombre de bras
        epsilon : probabilité d'exploration
        alpha   : taux d'apprentissage (0 < alpha <= 1)
        """
        self.K = K
        self.epsilon = float(epsilon)
        self.alpha = float(alpha)
        # Estimations de la récompense moyenne Q_k
        self.Q = np.zeros(K, dtype=float)
        self.rng = np.random.default_rng() if rng is None else rng

    def select_action(self) -> int:
        """
        Choisit une action selon la stratégie epsilon-glouton :
        - avec probabilité epsilon : action aléatoire (exploration)
        - sinon                    : action argmax_k Q_k (exploitation)
        """
        # TODO :
        # - tirer un nombre u dans [0, 1[
        # - si u < epsilon : renvoyer une action aléatoire
        # - sinon : renvoyer l'indice du max de Q
        raise NotImplementedError

    def update(self, action: int, reward: float) -> None:
        """
        Met à jour l'estimation Q[action] à partir de la récompense observée.
        Cette mise à jour est une moyenne mobile exponentielle où :
        - alpha contrôle le poids des nouvelles observations
        - Si alpha = 1 : seule la dernière récompense compte
        - Si alpha petit : l'historique a plus de poids
        - Contrairement à la moyenne arithmétique, cela permet de suivre
          un environnement non-stationnaire.
        Q[action] <- Q[action] + alpha * (reward - Q[action])
        """
        # TODO : implémenter la mise à jour de Q[action]

        raise NotImplementedError
```

### 2.2. Simulation d'un épisode

Dans le fichier « `src/main_bandit.py` », ajoutez une fonction :

```python
from agent import EpsilonGreedyAgent

def run_episode(env, epsilon: float, alpha: float, T: int, rng=None):
    """
    Simule un épisode de T pas de temps avec un agent epsilon-glouton.
    Renvoie la récompense moyenne cumulée (array de taille T).
    """
    if rng is None:
        rng = np.random.default_rng()

    agent = EpsilonGreedyAgent(env.K, epsilon=epsilon, alpha=alpha, rng=rng)

    rewards = []

    # TODO :
    # pour t de 0 à T-1 :
    #   - choisir une action avec agent.select_action()
    #   - jouer cette action dans l'environnement : env.step(action, t)
    #   - appeler agent.update(action, reward)
    #   - stocker la récompense

    rewards = np.array(rewards)
    cumulative_mean = np.cumsum(rewards) / np.arange(1, T + 1)
    return cumulative_mean
```

Modifiez ensuite la fonction `main()` de façon à :

— commenter l'appel à la stratégie aléatoire;
— lancer un épisode $\epsilon$-glouton avec par exemple `epsilon = 0.1` et `alpha = 0.05`;
— tracer la récompense moyenne cumulée.

**Questions** :

— La courbe $\epsilon$-glouton est-elle meilleure que celle obtenue pour une stratégie aléatoire dans l'exercice n°1 ?
— L'agent semble-t-il s'adapter aux changements de l'environnement ?

## Exercice n°3 : Influence des paramètres $\epsilon$ et $\alpha$

On s'intéresse à présent à l'influence des paramètres $\epsilon$ (« quantité d'exploration ») et $\alpha$ (« vitesse d'adaptation »).

### 3.1. Moyenne sur plusieurs épisodes

Dans le fichier « `src/main_bandit.py` », ajoutez la fonction :

```python
def run_multiple_episodes(env_K, epsilon, alpha, T, n_episodes=20):
    """
```

```
    Lance n_episodes indépendants pour un couple (epsilon, alpha),
    et renvoie la courbe de récompense moyenne cumulée moyenne
    sur ces épisodes.
    """
    all_cum_means = []

    for seed in range(n_episodes):
        rng = np.random.default_rng(seed=seed)
        env = KArmedBandit(env_K, rng=rng)
        cum_mean = run_episode(env, epsilon=epsilon, alpha=alpha, T=T, rng=rng)
        all_cum_means.append(cum_mean)

    all_cum_means = np.stack(all_cum_means, axis=0)  # (n_episodes, T)
    mean_curve = all_cum_means.mean(axis=0)
    return mean_curve
```

#### 3.2. Comparaison visuelle

1. Dans la fonction « `main()` », choisissez par exemple :
   — `epsilon` dans { 0.0, 0.05, 0.1, 0.3 }, avec `alpha` fixé (par exemple 0.05) ;
   — `alpha` dans { 0.01, 0.05, 0.15 }, avec `epsilon` fixé (par exemple 0.1).
2. Pour chaque valeur, calculez une courbe moyenne, puis tracez-les sur un même graphique. Indication : vous pouvez créer une fonction utilitaire qui prend une liste de courbes et de labels et les affiche ensemble.

**Questions** :

— Que se passe-t-il quand `epsilon` = 0 (aucune exploration) ?
— Que se passe-t-il quand `epsilon` est trop grand ?
— Comment `alpha` influence-t-il la réactivité de l'agent aux changements ?

## Exercice n°4 (bonus) : Qualité du bras choisi

Pour visualiser plus finement le comportement de l'agent, on peut tracer, à chaque pas de temps $t$, la probabilité réelle de gain du bras choisi.

Pour cela on réalise :

— pendant la simulation, stocker aussi les actions choisies ;
— après coup, pour chaque $t$, récupérer `env.true_probs(t)[action[t]]`;
— tracer cette quantité en fonction de $t$.

**Questions** :

— L'agent suit-il rapidement les bras redevenus intéressants ?
— Que se passe-t-il si `alpha` est trop petit ? trop grand ?

## Exercice n°5 (bonus) : Mesurer la performance de l'agent

Jusqu'ici, vous avez surtout observé la récompense moyenne cumulée par l'agent. C'est une mesure utile, mais elle ne dit pas clairement à quel point l'agent se rapproche d'un comportement optimal.

Dans cet exercice, vous allez introduire deux mesures plus fines :

— le « regret dynamique » par rapport à un « oracle »;
— la qualité du bras choisi au cours du temps.

On supposera que vous avez déjà une simulation qui, à chaque pas de temps $t$ :

— choisit une action `a_t` avec l'agent;
— obtient une récompense `r_t = env.step(a_t, t)`;
— met à jour l'agent avec `agent.update(a_t, r_t)`.

Vous modifierez la fonction `run_episode()` (ou en créerez une variante) afin de calculer ces grandeurs.

### 5.1. Regret dynamique

L'environnement met à disposition les probabilités réelles :

```
p_t = env.true_probs(t)   # array de taille K
```

Le meilleur bras à l'instant $t$ est celui qui maximise cette probabilité :

$$p_t^\star = \max_k p_t[k].$$

Un oracle qui connaît ces probabilités jouerait toujours ce meilleur bras. Son espérance de récompense cumulée sur $T$ pas de temps est :

$$R_T^\star = \sum_{t=0}^{T-1} p_t^\star.$$

Pour votre agent, vous observez des récompenses réelles $r_t \in \{0, 1\}$. On définit alors le « regret dynamique » empirique de la manière suivante :

$$\hat{R}_T^{\text{dyn}} = \sum_{t=0}^{T-1} (p_t^\star - r_t).$$

Plus ce regret est faible, plus l'agent se comporte comme l'oracle.

1. Modifiez votre fonction « `run_episode()` » pour calculer, en plus de la récompense moyenne cumulée, le regret dynamique cumulé au cours du temps. Indication :

```
regrets = []
regret_cum = 0.0

for t in range(T):
    p_t = env.true_probs(t)  # probabilités réelles à l'instant t
    p_star_t = np.max(p_t)   # probabilité du meilleur bras

    # ...
    # choisir a_t, obtenir r_t, mettre à jour l'agent
    # ...

    regret_cum += (p_star_t - r_t)
    regrets.append(regret_cum)
```

2. Tracez sur un même graphique :
   — la récompense moyenne cumulée ;
   — le regret dynamique moyen ($\hat{R}_t^{\text{dyn}}/t$) (par exemple en divisant `regrets` par `np.arange(1, T+1)`).
3. Comparez ces courbes pour différents couples $(\epsilon, \alpha)$ vus à l'exercice n°3.

**Questions** :

— Un « meilleur » agent doit-il avoir un regret croissant lentement ou rapidement ?
— Qu'observez-vous lorsque $(\epsilon = 0)$ (pure exploitation) ?
— Comment $\alpha$ influence-t-il la vitesse à laquelle le regret se stabilise ?

### 5.2. Qualité du bras choisi

Le regret dynamique compare l'agent à un oracle. On peut aussi suivre directement la qualité du bras choisi à l'instant $t$ :

$$q_t = p_{a_t}(t),$$

où $(a_t)$ est l'action choisie et $p_{a_t}(t)$ la probabilité réelle de gain du bras joué.

On peut également définir l'écart au meilleur bras :

$$\Delta_t = p_t^\star - q_t.$$

1. Modifiez votre boucle de simulation pour enregistrer les actions `a_t` et calculer, à chaque pas `t`, la quantité `q_t` et éventuellement `Delta_t`. Esquisse :

```
qualities = []
gaps = []

for t in range(T):
    p_t = env.true_probs(t)
```

```
    p_star_t = np.max(p_t)

    # choisir l'action
    a_t = agent.select_action()
    r_t = env.step(a_t, t)
    agent.update(a_t, r_t)

    q_t = p_t[a_t]
    qualities.append(q_t)
    gaps.append(p_star_t - q_t)
```

2. Tracez qualities en fonction du temps $t$. Vous pouvez également tracer une moyenne glissante de `gaps` pour lisser les courbes (par exemple avec une fenêtre de 50 pas).

**Questions** :

— Quand l'environnement change (probabilités qui montent/descendent), l'agent suit-il rapidement les nouveaux bras intéressants ?

— Que se passe-t-il si $\alpha$ est trop petit ? Si $\alpha$ est trop grand ?

— Comment ces observations complètent-elles l'information donnée par la seule récompense moyenne cumulée ?

### 5.3. Taux de sélection du bras optimal

On peut enfin mesurer la fraction de temps où l'agent joue un bras optimal ou quasi-optimal.

1. Pendant la simulation, comptez le nombre de fois où l'action choisie fait partie des bras maximisant $p_t$. Vous pouvez aussi considérer un bras « presque optimal » s'il vérifie $p_{a_t}(t) \geq p_t^\star - \delta$ pour un petit $\delta > 0$ (par exemple $\delta = 0.05$).
2. Calculez le taux :
$$\text{taux\_opt} = \frac{1}{T} \sum_{t=0}^{T-1} \mathbf{1}\{a_t \in \arg\max_k p_k(t)\}$$
3. Comparez ce taux pour différentes valeurs de $\epsilon$ et $\alpha$.

**Questions** :

— Quel couple $(\epsilon, \alpha)$ donne le meilleur compromis entre exploration et exploitation dans cet environnement ?

— Ces mesures (regret, qualité du bras choisi, taux de bras optimal) donnent-elles toutes les mêmes conclusions, ou mettent-elles en avant des aspects différents du comportement de l'agent ?

**Pour aller plus loin**
Les notions de valeur d'action $Q$, de stratégie $\epsilon$-glouton et plus généralement l'apprentissage par renforcement sont présentées de manière approfondie dans l'ouvrage de référence suivant :
R. S. Sutton, A. G. Barto, *Reinforcement Learning : An Introduction*, MIT Press, 2e édition, 2018.

Les chapitres 2 et 6 ont d'ailleurs directement inspirés ce TP.

## Conclusion

Dans ce TP, vous avez :

— simulé un bandit-manchot à $K$ bras ;
— mis en oeuvre un agent de type $\epsilon$-glouton ;
— étudié l'influence de $\epsilon$ (exploration) et de $\alpha$ (adaptation) ;
— visualisé l'apprentissage dans un environnement instable.

Ce cadre simple prépare aux méthodes de renforcement plus avancées telles que le *Q-learning* ou encore SARSA (*State-Action-Reward-State-Action*).