Ejercicios del Capítulo 2.

Ej. 2.1.

Los dos métodos ε-greedy tenderán a encontrar el valor real de $q(a)$:

$$\lim_{t \to \infty} Q(a) = q(a)$$

Por tanto, el comportamiento a largo plazo de ambos será el mismo (no siendo este el caso del algoritmo sin exploración, que puede quedar atascado en un valor de Q y no seguir mejorando).

El método con $\varepsilon = 0.1$ usará a la larga la acción óptima un 90% de las veces, mientras que el de $\varepsilon = 0.01$ lo hará un 99% de las veces, lo que supone una mejora del 10%.

Ej. 2.2.

Habíamos llegado a una expresión como

$$Q_{n+1} = \alpha_n R_n + \alpha_{n-1}(1-\alpha_n) R_{n-1} + (1-\alpha_n)(1-\alpha_{n-1}) Q_{n-1}$$

Podemos extenderlo fácilmente un paso más:

$$Q_{n+1} = \alpha_n R_n + \alpha_{n-1}(1-\alpha_n) R_{n-1} + \alpha_{n-2}(1-\alpha_{n-1})(1-\alpha_{n-2}) R_{n-2} +$$

$$+ (1-\alpha_n)(1-\alpha_{n-1})(1-\alpha_{n-2}) Q_{n-2}$$

El término en R_i queda $\sum_{i}^{n} \alpha_i R_i \prod_{k=i+1}^{n} (1-\alpha_k)$

Y el término en Q:

$$\prod_{i=1}^{n} (1-\alpha_i) Q_1$$

Y por tanto:

$$Q_{n+1} = \prod_{i=1}^{n} (1-\alpha_i) Q_1 + \sum_{i}^{n} \alpha_i R_i \prod_{k=i+1}^{n} (1-\alpha_k)$$

Ej. 2.5

~~Durante la primera parte~~,

$$Q_{K+1} = Q_K + \alpha [R_K - Q_K]$$

Si ponemos de Q_1 mucho más alto que las demás la media de recompensas que x va a obtener, tardaremos en converger a su valor de Q_K: ← probablemente esto sea bajo

$$Q_2 = Q_1 + \alpha [R_1 - Q_1]$$

$$A_{t+2} = \arg\max_a Q_t(a)$$

El valor seleccionado de Q_1 y, concretamente, lo cerca que α esté de $\bar{R}$ que sea, determinará si este método converge rápido o lento.

Suponemos Q_1 fijo, con $Q_1 = 5$.

~~Si α es pequeño~~

Supongamos que calculamos Q_2 de $\{a_1, \ldots, a_n\}$ y encontramos la acción óptima.

En ese punto, el algoritmo actuará de forma "greedy" y seguirá seleccionándola, convergiendo a su valor real (probablemente menor que 5). En ese punto, el resto de acciones tendrán aún un valor estimado superior a 5, ~~y entonces~~ (debido a falta de conocimiento del algoritmo) por lo que el agente cambiará de acción a una subóptima.

Ej. 2.6.

Caso A: $\begin{cases} a_1 : 0.1\ (50\%) \\ a_2 : 0.2\ (50\%) \end{cases}$

Caso B: $\begin{cases} a_1 : 0.9\ (50\%) \\ a_2 : 0.8\ (50\%) \end{cases}$

No sé en qué caso estoy. $E[R]$? ¿Cómo debo comportarme?

Si estoy en el caso A, lo mejor que puedo hacer es escoger a_2.

" " " " " B, " " " " " " " a_1.

Si juego 100 veces seleccionando "a_1", tendré 0.05 + 0.45 = 0'5

" " " " " "a_2". " 0.01 + 0.4 = 0'5

¿Y si selecciono al azar?

$$0.5 \cdot [0'5 \cdot 0'1 + 0'5 \cdot 0'2] + 0'5 [0'5 \cdot 0'9 + 0'5 \cdot 0'8] =$$

$$= \frac{1}{2} [0'15] + \frac{1}{2} [0.45 + 0.4] = \frac{1}{2} 0'15 + \frac{1}{2} 0.85 =$$

$$= 0.075 + 0.425 = 0'5$$

Por tanto, da igual lo que haga (siempre a_1, siempre a_2 o $\frac{1}{2}a_1 + \frac{1}{2}a_2$), siempre obtengo $E[R]$ de 0'5.

Imaginemos ahora que sabemos si estamos en A o B.

Si estoy en A, debo escoger a2. Si estoy en B, debo escoger a4.

Espero tener $0'5 \cdot 0'2 + 0'5 \cdot 0'9 = 0.1 + 0.45 = 0'55$

Ej. 2.2.
n-bandit
*
$\alpha = \frac{1}{k} \cdot \frac{1}{n}$

Greedy.

bandit(a) = R

Usar arrays y variables, nada de subscripts.

Recordemos:

$$Q_{n+1} = Q_n + \alpha (R_n - Q_n).$$

$$A_n = \arg\max_a Q_n(a)$$

Variables que guardo: Q, n, $\vec{A}$

~~aux~~ old ← -1000

Q ← 0

n ← 1

for ~~u~~ in [1, 2000]:
  for a in A:
    if $Q^{[a]} + \alpha(\text{bandit}(a) - Q^{[a]}) >$ ~~aux~~ old
      ~~aux~~ old ← $Q^{[a]} + \alpha(\text{bandit}(a) - Q^{[a]})$

  $Q^{[a]}$ ← ~~aux~~ old

~~aux~~ old = -1000

Ej. 2.4: Diseñe un ej. para demostrar las dificultades a las que se enfrentan los ~~~~ métodos average-sample en problemas no estacionarios.

Usar 10-armed bandit, con $q(a)$ comenzando igual y después siendo independientes (random walks).

Usar $\alpha = \frac{1}{k}$ y otro con $\alpha = 0.1$

Usar $\varepsilon = 0.1$

Necesito:
- Una función que genere los valores $q(a_1), \dots, q(a_{10})$ en cada instante
- Una función que genere el nido.
- bandit(a) (combina ambas).
- Una función que escoja la acción → greedy / pero con ε-exploración

El pseudocódigo es → $Q_{n+1} = Q_n + \alpha (R_n - Q_n)$ $(\alpha = \frac{1}{k})$ (Es decir, como en el problema 2.2).

He hecho un programa que se comporte de manera similar a las curvas del libro. No obstante, a continuación transcribo y desmenuzo el esquema seguido por Neuromatch Academy (mirar carpeta Doctorado > RL):

1º Definim una funció que seleccioni la acció de m a per valor amb prob. $1-\varepsilon$, i una acció aleatòria amb prob. ε.

```
def epsilon_greedy (q, epsilon):
    Args:
      q (ndarray): array de valores de los n-acciones.
      epsilon (float): prob. de explorar.

    si random > epsilon:
          seleccionar  a = argmax (q)

    else:
          a = seleccionar aleatoriamente ⇒ a = np.random.choice(len

    return a
```

```
def update_action_value (q, action, reward, alpha):

    Args:
      q (ndarray).
      action (int)
      reward (float)
      alpha (float)

    Returns:
      float: the updated value for the selected action.
```

value = q[action] + α · (r − q[action])

return value.

Finalmente, el programa ppal:

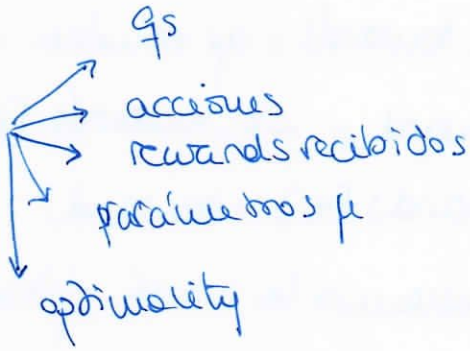
def multi_armed_bandit(n_arms, epsilon, alpha, n_steps):

Args:

n_arms (int)
epsilon (float)
alpha (float)
n_steps (int)

Returns:

dict: un diccionario con



#1: las acciones tienen valores reales q(a;) provenientes de una N(0,1):

μ = np.random.normal(size = n_arms) → vector $\vec{q}(\vec{a})$.

los valores calculados $\vec{Q}_t(\vec{a})$ se inicializan a ϕ.
Guardan los Q_t de cada iteración para analizar.
También guarda los rewards observados (uno por iteración),
las acciones seleccionadas (una por iteración) y el # nº de veces que se escoge una óptima.

```
# inicialmente todo es 0:

q = np.zeros(n_arms)
qt = np.zeros((n_steps, n_arms))
rewards = np.zeros(n_steps)
actions = np.zeros(n_steps)
optimal = np.zeros(n_steps)

#Run the bandit
for t in range(n_steps):

    action = epsilon_greedy(q, epsilon)
    actions[t] = action

    all_rewards = np.random.normal(mu)   ← premios obtenidos. Tienen media μ y variante 1
    reward = all_rewards[action]
    rewards[t] = reward

    # buscamos la acción óptima y comprobamos si coincide
    # con action
    optimal_action = np.argmax[all_rewards]
    optimal[t] = action == optimal_action

    #Update the action value
    q[action] = update_action_value(q, action, reward, alpha]

    qt[t] = q

results = { 'qt' = qt,
            'actions' = actions,
            'rewards' = rewards,
            'optimal' = optimal,
          }
```