Trabajo Práctico 2:
Deep Q-Network

Lenguajes permitidos: Python 3
Librerías permitidas: PyTorch, Gymnasium ≥ 1.0.0, NumPy, tqdm
Entrega: A través del campus virtual — Informe (máx. 3 páginas en PDF) y código fuente (comprimido en un archivo .zip)

Objetivo

El objetivo principal de este trabajo práctico es implementar el algoritmo DQN y emplearlo para entrenar agentes en distintos entornos. Como introducción al tema, se solicitará previamente una implementación del algoritmo Q-learning. El trabajo se divide en cuatro partes:

Parte 1 — Entornos de Prueba

Para validar la correcta implementación del agente y su función de valor, se proponen tres entornos artificiales simples, diferentes a los de TP1, cada uno diseñado para aislar aspectos específicos del aprendizaje y permitir una verificación paso a paso del funcionamiento del algoritmo. Cada entorno debe implementarse como una subclase de *gymnasium.Env*. Se utilizará un factor de descuento γ=0.99 en todos los casos.

ConstantRewardEnv

Acción única, sin observación, un solo paso, recompensa constante

- **Acciones disponibles**: 1 (única acción)
- **Observaciones**: constante 0
- **Duración**: 1 paso de tiempo (episodio de una sola transición)
- **Recompensa**: +1 en cada episodio

Este entorno permite verificar si la función de valor aprende correctamente un valor constante. Dado que la observación es siempre la misma y la recompensa también, el agente debería rápidamente aprender que el valor esperado es 1. Si no lo hace, el problema podría estar en el cálculo de la función de pérdida de valor o en la configuración del optimizador.

RandomObsBinaryRewardEnv

Acción única, observación aleatoria, un solo paso, recompensa dependiente de la observación

- **Acciones disponibles:** 1 (única acción)
- **Observaciones:** aleatorias, con valor +1 o -1
- **Duración:** 1 paso de tiempo
- **Recompensa:** coincide con la observación (+1 o -1)

Este entorno permite evaluar si el agente es capaz de aprender un valor que depende de la observación. A diferencia del entorno anterior, la red debe aprender a predecir un valor condicional a la entrada. Si el agente aprendió correctamente en el entorno **ConstantRewardEnv** pero falla aquí, puede indicar un problema en el flujo de gradientes o en la retropropagación dentro de la red neuronal.

TwoStepDelayedRewardEnv

Acción única, observación determinista, dos pasos, recompensa diferida

- **Acciones disponibles:** 1 (única acción)
- **Observaciones:** en el primer paso se observa 0; en el segundo paso se observa 1
- **Duración:** 2 pasos por episodio
- **Recompensa:** 0 en el primer paso, +1 al final del episodio

Este entorno introduce un componente temporal: la recompensa no es inmediata, sino que se recibe al final del episodio. El valor correcto debe descontar adecuadamente la recompensa futura. Si el agente aprende correctamente en el entorno RandomObsBinaryRewardEnv pero falla en este, el problema puede estar en la implementación del mecanismo de descuento o en el manejo de la acumulación de recompensas a lo largo del episodio.

Parte 2 — Implementación del Algoritmo Q-Learning

Q-learning es un algoritmo de aprendizaje por refuerzo ***model-free*** dado que permite aprender una política óptima para un agente cuya probabilidad de transición (dinámica) es desconocida, siempre y cuando puede ser modelado como un proceso de decisión de Markov **(MDP**). Su objetivo es aprender una función **Q(s,a)** que estime el valor esperado de realizar una acción **a** en un estado **s**, y seguir luego una política que maximice dicha función.

Durante la interacción con el entorno, el agente observa transiciones de la forma **(s,a,r,s′)** y actualiza su estimación de **Q(s,a)**. El pseudocódigo es el siguiente:

Algorithm 1 Q-learning Pseudocode

Initialize $Q(s, a)$ for all states s and actions a, to zero.
Initialize hyperparameters: learning rate $\alpha \in (0, 1]$, discount factor $\gamma \in [0, 1]$, and exploration rate $\epsilon \in [0, 1]$.
repeat for each episode:
  Initialize the agent's state s by sampling from the set of possible initial states.
  repeat for each step of the episode:
    Choose action a from state s using an ϵ-greedy policy:
      With probability ϵ, choose a random action a
      With probability $1 - \epsilon$, choose $a = \arg\max_{a'} Q(s, a')$
    Take action a, observe reward r and new state s'
    Update the Q-value for the state-action pair (s, a):
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    Update the current state to the new state: $s \leftarrow s'$
  until s is a terminal state or termination criteria is met (e.g., a fixed number of steps)
until termination criterion is met (e.g., a fixed number of episodes, or convergence)

Se pide:

1. Implementar el algoritmo 1 y aplicarlo para entrenar a un agente que aprenda a resolver el entorno **FrozenLake-v1**, utilizando una grilla 4x4, con una tabla Q[s,a] y su correspondiente regla de actualización tal como se indica en el pseudocódigo.

2. Entrenar al agente durante una cantidad fija de episodios (por ejemplo, 5.000). Experimentar con diferentes hiperparametros y graficar los rewards, hacer una comparacion.
3. Evaluar el desempeño del agente después del entrenamiento:
    a. Porcentaje de éxito en 100 episodios.
    b. Porcentaje de éxito en 100 episodios con exploración fija (por ej, ε = 0.1).
4. Entrenar con una política de exploración epsilon-**greedy** con opción de decaimiento de ε.

```
min_epsilon + (max_epsilon-min_epsilon)*np.exp(-decay_rate*episode)
```

5. Para el punto anterior, producir un gráfico de evolución de la recompensa promedio cada N episodios (por ejemplo, cada 100)

Parte 3 — Implementación del Algoritmo DQN

Antes de comenzar a escribir código, les recomendamos enfáticamente que lean y comprendan el artículo sobre DQN.

El algoritmo **Deep Q-Network (DQN)** utiliza redes neuronales profundas para resolver problemas de decisión secuencial en entornos complejos. Su objetivo es aprender una política que maximice la recompensa esperada acumulada, aproximando la **función Q** con una red neuronal en lugar de una tabla explícita, como en el Q-learning tradicional.

Para lograr una estabilidad adecuada durante el entrenamiento, DQN incorpora dos técnicas clave: **replay buffer**, que almacena experiencias pasadas para romper la correlación temporal de los datos, y **target networks**, que estabilizan las actualizaciones del valor Q.

Algorithm 2: Deep Q-learning con repetición de experiencias

Input: Capacidad del buffer N, número máximo de episodios M, número máximo de pasos T
Output: Función de valor-acción entrenada Q

Inicializar buffer D con capacidad N;
Inicializar función de valor-acción Q con pesos aleatorios θ;
Inicializar función de valor-acción objetivo Q' con pesos $\theta' = \theta$;
for *episodio* $= 1, \ldots, M$ **do**
  Inicializar estado s_1;
  for $t = 1, \ldots, T$ **do**
    Con probabilidad ε seleccionar una acción aleatoria a_t;
    En otro caso seleccionar $a_t = \arg\max_a Q(s_t, a; \theta)$;
    Ejecutar acción a_t en el entorno y observar la recompensa r_t y estado próximo s_{t+1};
    Almacenar la transición (s_t, a_t, r_t, s_{t+1}) en D;
    Muestrear un minibatch aleatorio de K transiciones (s_j, a_j, r_j, s_{j+1}) desde D;
    Definir

$$y_j = \begin{cases} r_j & \text{si el episodio termina en el paso } j+1 \\ r_j + \gamma \max_{a'} Q'(s_{j+1}, a'; \theta') & \text{en otro caso} \end{cases}$$

    Realizar un paso de descenso de gradiente sobre

$$(y_j - Q(s_j, a_j; \theta))^2$$

    con respecto a los parámetros de la red θ;
    Cada C pasos actualizar $Q' = Q$;
  end
end

3.1 Implementación sobre CartPole-v1

- Se pide implementar el **Algoritmo 2 con exploración epsilon-greedy con opción de decaimiento de ε** para entrenar un agente que aprenda a resolver el entorno CartPole-v1.
    - La política del agente debe estar representada mediante una red neuronal. Se sugiere comenzar con la siguiente arquitectura base:

      nn.Linear(input, 64),
      nn.RELU(),
      nn.Linear(64, 64),
      nn.RELU(),
      nn.Linear(64, output)

      Sugerencia: Pueden experimentar con otras configuraciones de capas (más profundas o más anchas) y distintas funciones de activación (por ejemplo, Tanh, ELU o LeakyReLU) para observar cómo influyen en la velocidad y estabilidad del aprendizaje.

- Entrenar al agente durante una cantidad fija de episodios (por ejemplo, 500 o 1000). Considerar el preprocesado como la función identidad.
- Visualizar el desempeño del entrenamiento utilizando **TensorBoard**, incluyendo al menos:
    - La evolución de la **función de pérdida (loss)**.
    - La **recompensa promedio por episodio**

- Entrenar hasta que se considere el entorno ***resuelto*** (esto es cuando el reward converge y se está logrando el objetivo del entorno, en el caso de *Cartpole* que se mantenga el péndulo durante todo el episodio ~1000 steps).

3.2 Implementación sobre MinAtar/Breakout-v0

- Se pide implementar el **Algoritmo 2 con exploración epsilon-greedy con opción de decaimiento de ε** para entrenar un agente que aprenda a resolver el entorno MinAtar/Breakout-v0.
    - La política del agente debe estar representada mediante una red neuronal. Se sugiere comenzar con la arquitectura recomendada en el artículo de DQN, pero adaptarla al entorno de MinAtar.
    - **Sugerencia**: Pueden experimentar con otras configuraciones de capas (más profundas o más anchas), distintos parámetros de las redes y distintas funciones de activación (por ejemplo, Tanh, ELU o LeakyReLU) para observar cómo influyen en la velocidad y estabilidad del aprendizaje.
- Entrenar al agente durante una cantidad fija de episodios (por ejemplo, 500 o 1000). Leer atentamente cómo está representada la información en el repositorio.
    - Para poder correr correctamente MinAtar con Gymnasium, además de las instrucciones en el repositorio, hay que descargar el repo e importar la función *registers_envs* de *minatar.gym* y ejecutarla para poder registrar el ambiente y tenerlo disponible para Gymnasium.
- Visualizar el desempeño del entrenamiento utilizando **TensorBoard**, incluyendo al menos:
    - La evolución de la **función de pérdida (loss)**.
    - La **recompensa promedio por episodio**
- Entrenar hasta que se considere el entorno ***resuelto*** (esto es cuando el reward converge y se está logrando el objetivo del entorno, en el caso de *Breakout* que logre terminar el nivel sin perder 3 vidas consecutivas).

Parte 4 — Experimentos y Evaluación

- Grafique curvas de recompensa vs episodios para DQN vs el REINFORCE del tp pasado para el ambiente **CartPole-v1**
- Grafique curvas de recompensa vs *wall time* para DQN vs el REINFORCE del tp pasado para el ambiente **CartPole-v1**

Entrega

La entrega se realiza a través del campus virtual, e incluye:

Código fuente

Todo el código utilizado para definir entornos, entrenar agentes y generar las figuras. Debe ser original y estar comprimido en un archivo .zip.
No se evaluará el estilo ni la legibilidad del código, pero se utilizará para comprobar errores de implementación y verificar la autoría de dicho código.

Informe:

Un archivo PDF de **máximo 3 páginas**, que debe incluir todos los gráficos y responder de manera muy **sintética y clara** las siguientes preguntas:

1. Se incluyen las curvas de recompensa media por episodio para cada uno de los tres entornos (para ambos algoritmos):
    a. ConstantRewardEnv
    b. RandomObsBinaryRewardEnv
    c. TwoStepDelayedRewardEnv
2. ¿Qué ventajas y desventajas observó entre el método de Policy Gradient (REINFORCE) y DQN? Considerar aspectos como: estabilidad, velocidad de convergencia, sensibilidad a hiperparámetros, variabilidad de resultados, uso de memoria, etc.
3. ¿Cuál de los dos métodos logró una mejor performance en **CartPole-v1** en sus experimentos? Justifique en base a las curvas de recompensa y comportamiento observado. Incluya los gráficos de la sección 4.
4. Para el caso de **MinAtar/Breakout-v0**: ¿Qué impacto tuvo el uso del *replay buffer* y la *target network* en DQN? ¿Qué ocurre si se omite alguno de estos mecanismos? Experimente y justifique.

5. ¿Qué dificultades encontró al implementar y entrenar cada uno de los algoritmos? Pueden referirse a temas como: debugging, elección de hiperparámetros, uso de TensorBoard, etc.

Criterio de aprobación

El trabajo se considera aprobado si se completa hasta la Parte 3.1