UNIVERSIDAD MAYOR REAL Y PONTIFICIA
DE SAN FRANCICO XAVIER

FACULTAD DE CIENCIAS Y TECNOLOGÍA

Laboratorio 7

Aprendizaje por Refuerzo

Universitario (a): Zurita Paco Elvis Jherson
Escobar Ruiz Marco Antonio

Carrera: Ingeniería de Sistemas

Docente: Pacheco Lora Carlos Walter

Materia: INTELIGENCIA ARTIFICIAL I

Fecha: 10/06/2025

Implementación de cada Técnica

Q-Learning Estándar: Los Fundamentos

Objetivo:

Entrenar a un agente para aprender la mejor secuencia de acciones en un entorno (como un laberinto) con el objetivo de maximizar sus recompensas totales, utilizando el algoritmo Q-Learning en su forma más común.

---

1. La Tabla Q: La Memoria del Agente

- ¿Qué es? Una tabla (q_table) que guarda la "calidad" (Q-value) de tomar una acción específica (a) desde un estado particular (s).
- Inicialización (np.zeros(...)):
    - q_table = defaultdict(lambda: np.zeros(mo_gym.make(env_name).action_space.n))
    - Todos los Q-values se inician en cero. Esto significa que al principio, el agente no tiene ninguna expectativa de recompensa para ninguna acción en ningún estado.
- Propósito: La tabla Q es el "mapa de valor" que el agente construye a medida que aprende. Un Q-value alto para un par (estado, acción) indica una buena estrategia.

---

2. Estrategia de Selección de Acción: ϵ-greedy

- action = epsilon_greedy(q_table[state], epsilon)
- Esta política es el equilibrio entre explorar y explotar:
    - Exploración: Con una pequeña probabilidad (ϵ, por ejemplo, 0.1 o 10%), el agente elige una acción completamente aleatoria. Esto le permite descubrir nuevas rutas y experiencias que de otra forma podría no encontrar.
    - Explotación: Con la probabilidad restante (1−ϵ, por ejemplo, 0.9 o 90%), el agente elige la acción que, según su tabla Q, tiene el Q-value más alto en el estado actual. Esto aprovecha lo que ya ha aprendido para maximizar las recompensas.
- epsilon=0.1: Un valor típico que asegura que el agente explore lo suficiente sin ignorar lo que ya sabe.

---

3. La Fórmula de Actualización de Q-learning: El Corazón del Aprendizaje

Cada vez que el agente realiza una acción, recibe una recompensa y transiciona a un nuevo estado. La tabla Q se actualiza usando la siguiente ecuación:

$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R + \gamma \cdot a' \max Q(s',a') - Q(s,a))$

- Q(s,a): El valor Q actual del par (estado, acción) que acaba de tomarse.
- α (alpha): Tasa de Aprendizaje (0.1)
    - Determina cuánto de la nueva información (TD Error) se incorpora al Q-value existente. Un α=0.1 significa que el 10% del error se "aprende" en cada paso.
- R (scalar_reward): Recompensa Inmediata
    - La recompensa que el agente obtuvo directamente por tomar la acción a en el estado s.
- γ (gamma): Factor de Descuento (0.99)
    - Valora las recompensas futuras. Un γ cercano a 1 (0.99) significa que el agente considera las recompensas futuras casi tan importantes como las inmediatas, fomentando una estrategia a largo plazo.
- $\max_{a'} Q(s',a')$ (best_next): Valor del Siguiente Estado
    - El máximo Q-value posible desde el estado siguiente s'. Esto es clave: el agente aprende no solo de la recompensa inmediata, sino también de lo bueno que es el *siguiente estado* si se toma la mejor acción desde allí.
- $(R + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a))$: Error TD (Temporal Difference Error)
    - La diferencia entre lo que el agente esperaba (Q(s, a)) y la "verdad" mejorada que acaba de experimentar o prever (td_target). Este error es el que guía el aprendizaje.

El Código: Un Bucle de Aprendizaje

Python

```
# Reinicia el entorno y obtiene el estado inicial
obs, _ = env.reset()
state = tuple(obs)

while True:
    # Selecciona acción usando política ε-greedy
    action = epsilon_greedy(q_table[state], epsilon)

    # Ejecuta la acción en el entorno
    next_obs, reward, terminated, truncated, _ = env.step(action)
    scalar_reward = get_scalar_reward(reward) # Convierte la recompensa si es necesaria
```

```
# Si el episodio termina, rompe el bucle
if terminated or truncated:
    break

# Procesa la siguiente transición
next_state = tuple(next_obs)

# Actualiza la tabla Q usando la ecuación de Q-learning
best_next = np.max(q_table[next_state])
td_target = scalar_reward + gamma * best_next
q_table[state][action] += alpha * (td_target - q_table[state][action])

# Avanza al siguiente estado
state = next_state
```

Cómo se Aplica en el Entorno four-room-v0

Imaginemos nuestro agente en el four-room-v0 (un laberinto de cuatro habitaciones) buscando una meta.

1. Las Primeras Experiencias (Exploración al Azar)

- Inicio: El agente está en una casilla (un estado). Como su tabla Q está llena de ceros, no tiene preferencia. Gracias a ϵ-greedy, elige una acción casi al azar.
- Movimiento: El agente se mueve, el entorno le da una reward (cero o negativa por chocar con una pared) y lo lleva a un next_state.
- Actualización: La fórmula de Q-learning se aplica. Como el Q(s,a) era cero y las recompensas iniciales son bajas, los Q-values se mantendrán bajos o se harán negativos, enseñando al agente qué acciones *no* debe tomar.

2. Descubrimiento y Propagación de Valor

- **Recompensa**: Eventualmente, por suerte o por exploración, el agente alcanza la meta y recibe una recompensa positiva considerable (R).
- **Aprendizaje Hacia Atrás:**
    - El Q-value del par (estado, acción) que llevó directamente a la meta se actualizará con un valor positivo significativo.

- En episodios futuros, cuando el agente llegue a un estado s' que previamente condujo a un estado con un Q-value alto (gracias a maxa'Q(s',a')), el Q-value de las acciones que lo llevaron a ese s' también comenzará a aumentar.
- Este proceso se propaga desde la meta hacia atrás a lo largo de los caminos óptimos. Es como si el "valor" de la meta se fuera "contagiando" a las casillas que la preceden.

3. Convergencia y Política Óptima

- Después de muchos episodios (episodes=1000), el agente, a través de la exploración constante y la actualización iterativa, refinará los valores en su tabla Q.
- Con el tiempo, los Q-values de los caminos óptimos se diferenciarán claramente, y el agente tenderá a explotar estos caminos (elegir la acción con el Q-value más alto) la mayor parte del tiempo, llegando eficientemente a la meta.

En Resumen:

Este código implementa el Q-Learning clásico, un método fundamental en el aprendizaje por refuerzo. Permite que un agente aprenda a navegar y tomar decisiones óptimas en un entorno interactivo, construyendo su conocimiento de valor a través de la exploración gradual y la actualización de sus expectativas basada en las recompensas observadas.

Q-Learning con Tasa de Aprendizaje Decremental

Objetivo: Hacer que el agente confíe más en su conocimiento a medida que explora. La tasa de aprendizaje (α) disminuye cada vez que se repite un par (estado, acción).

---

1. El Concepto Clave: La Confianza se Gana con la Experiencia

- **Al principio:** Cuando el agente explora una ruta por primera vez, no sabe nada. Debe aprender mucho y rápido. Su tasa de aprendizaje (α) es alta.
- **Más tarde:** Después de pasar por la misma ruta muchas veces, el agente ya tiene una buena idea de su valor. No debería cambiar de opinión drásticamente por una sola nueva experiencia. Su tasa de aprendizaje (α) debe ser baja.

---

2. La Fórmula: Tasa de Aprendizaje Adaptativa

Para lograr esto, ajustamos la tasa de aprendizaje (α) con una fórmula que depende del número de visitas a un par (estado, acción), denotado como N(s,a).

$\alpha_{adaptativo} = \frac{\alpha_{base}}{1 + k \cdot N(s,a)}$

- α_{base}: La tasa de aprendizaje inicial (en el código, alpha).
- N(s,a): Cuántas veces hemos tomado la acción a desde el estado s.

- k: Una constante pequeña que controla la velocidad de la disminución (en el código, 0.01).

---

3. El Código: Implementación en Python

El código implementa esta idea de forma directa, manteniendo un contador de visitas.

Python

```
# 1. Llevar la cuenta de cuántas veces se ha tomado esta acción desde este estado
visit_count[state][action] += 1

# 2. Calcular la nueva tasa de aprendizaje adaptativa para esta actualización específica
adaptive_alpha = alpha / (1 + 0.01 * visit_count[state][action])

# 3. Usar esta tasa de aprendizaje reducida para actualizar la Tabla Q
#    (La actualización es la estándar de Q-Learning)
best_next = np.max(q_table[next_state])
td_target = scalar_reward + gamma * best_next
q_table[state][action] += adaptive_alpha * (td_target - q_table[state][action])
```

Explicación Oral: ¿Cómo se Aplican estas Fórmulas al Entorno four-room-v0?

(Esto es lo que explicarías verbalmente mientras muestras la diapositiva)

"Ahora, vamos a visualizar cómo funciona esto en nuestro laberinto de cuatro habitaciones. Piensen en nuestro agente como un explorador que va dibujando un mapa de 'calidad' en su libreta, que es nuestra Tabla Q.

Escenario 1: La Primera Visita a una Casilla

- Nuestro agente está en una casilla de inicio, el **estado s**. Decide moverse hacia arriba, la **acción a**.
- Esta es la **primera vez** que hace este movimiento desde esta casilla.
- En el código, visit_count[s][a] se incrementa y ahora vale 1.
- Cuando calculamos adaptive_alpha, el denominador (1 + 0.01 * 1) es muy cercano a 1. Por lo tanto, adaptive_alpha es casi igual a nuestra alpha base (por ejemplo, 0.1).
- **Conclusión:** El agente aprende con una tasa alta. La nueva información que obtiene tiene un gran impacto en su Tabla Q, porque es la primera vez que aprende algo sobre esta ruta.

Escenario 2: La Visita Número 100

- Ahora imaginemos que han pasado muchos episodios. El agente ha pasado por esa misma casilla de inicio y ha decidido moverse hacia arriba 99 veces antes. Esta es su visita número 100.
- visit_count[s][a] ahora vale 100.
- Calculamos adaptive_alpha: alpha / (1 + 0.01 * 100), que es alpha / (1 + 1), es decir, alpha / 2. ¡La tasa de aprendizaje se ha reducido a la mitad!
- Si el visit_count fuera 1000, la tasa de aprendizaje se dividiría por 11.
- **Conclusión:** La actualización que se hace a la Tabla Q es mucho más pequeña. El agente ya tiene una estimación muy estable del valor de moverse hacia arriba desde esa casilla. Una nueva experiencia no va a cambiar drásticamente su opinión. Es como decir: 'Ya he hecho esto 100 veces, estoy bastante seguro de lo que pasa. Este nuevo dato solo refinará un poco mi creencia'.

¿Por qué es esto útil?

Este método ayuda a que el aprendizaje **converja**. Evita que los valores de la Tabla Q sigan saltando erráticamente después de mucho entrenamiento. El agente primero explora y aprende de forma agresiva, y luego, a medida que gana confianza, sus conocimientos se estabilizan, llevando a una política final más robusta y fiable."

Q-Learning con Inicialización Optimista

Objetivo:

Entrenar a un agente para aprender la mejor política en un entorno (como un laberinto) maximizando las recompensas, con un sesgo inicial hacia la **exploración de lo desconocido**.

---

1. La Tabla Q: El Mapa de Valor del Agente

- **¿Qué es?** Un diccionario (q_table) que almacena el "valor de calidad" (Q-value) de tomar una **acción específica (a)** desde un **estado particular (s)**.
- **Significado:** Un Q-value alto para (s,a) sugiere que esa acción es buena para obtener recompensas futuras desde ese estado.
- **Inicialización Optimista (init_value=1.0):**
    - q_table = defaultdict(lambda: np.full(..., init_value))
    - Todos los valores Q se inician con un valor alto (por ejemplo, 1.0) para los estados y acciones no visitados.
    - **¿Por qué?** Anima al agente a probar acciones que no conoce, ya que "espera" que sean buenas. Esto fomenta una exploración más exhaustiva al principio.

---

2. Estrategia de Selección de Acción: Exploración vs. Explotación (epsilon_greedy)

- action = epsilon_greedy(q_table[state], epsilon)
- El agente debe equilibrar:
    - **Exploración:** Probar acciones nuevas para descubrir mejores caminos (ϵ probabilidad).
    - **Explotación:** Elegir la acción con el Q-value más alto conocido para maximizar la recompensa actual (1−ϵ probabilidad).
- **epsilon=0.1:** Un 10% de las veces, el agente toma una acción aleatoria para explorar.

---

3. La Fórmula de Actualización de Q-learning: Aprendiendo de la Experiencia

Cuando el agente toma una acción, el entorno le da una recompensa y un nuevo estado. La tabla Q se actualiza así:

$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R + \gamma \cdot a' \max Q(s',a') - Q(s,a))$

- **Q(s,a)**: El Q-value actual del par (estado, acción) que se acaba de tomar.
- **α (alpha): Tasa de Aprendizaje (0.1)**
    - Controla qué tan rápido el agente "cree" en la nueva información. Un α=0.1 significa que el 10% del error de aprendizaje se incorpora a la tabla Q.
- **R (scalar_reward): Recompensa Inmediata**
    - La recompensa que el agente obtiene directamente por esa acción.
- **γ (gamma): Factor de Descuento (0.99)**
    - Determina la importancia de las recompensas futuras. Un γ cercano a 1 (como 0.99) significa que las recompensas futuras son casi tan importantes como las inmediatas, fomentando el pensamiento a largo plazo.
- **maxa'Q(s',a') (best_next): Valor del Siguiente Estado**
    - Representa la "mejor recompensa futura esperada" desde el siguiente estado s'. El agente mira hacia adelante y considera el valor de la mejor acción que podría tomar desde allí.
- **(R+γ·maxa'Q(s',a')−Q(s,a)): Error TD (Temporal Difference Error)**
    - La diferencia entre lo que el agente esperaba (Q(s, a)) y lo que realmente obtuvo más lo que espera obtener en el futuro (td_target). ¡Este "error" es lo que impulsa el aprendizaje!

Cómo se Aplica en el Entorno four-room-v0

Imagina el entorno four-room-v0 como un laberinto sencillo donde el agente debe encontrar una meta (por ejemplo, una salida).

1. Las Primeras Exploraciones (Impulsadas por el Optimismo)

- **Inicio:** El agente se encuentra en una casilla del laberinto (un **estado**). Como todos los Q-values iniciales son altos (1.0), el agente está **dispuesto a probar cualquier dirección** (acción).
- **Movimiento:** Digamos que el agente se mueve "arriba". El entorno le da una reward (probablemente pequeña o cero por no haber llegado a la meta aún) y lo coloca en next_state.
- **Actualización (Fórmula):** La fórmula de Q-learning se aplica. Si la acción fue neutral en recompensa, pero el next_state aún tiene valores optimistas para sus acciones, la q_table[state][action] se actualizará ligeramente, pero el optimismo inicial seguirá siendo el principal motor. Si chocó con una pared, el reward negativo hará que el Q-value para esa acción en ese estado caiga drásticamente, aprendiendo rápido que es una mala idea.

2. Aprendizaje y Refinamiento (Iteraciones de la Fórmula)

- **Encuentro con la Meta:** Si, por pura exploración, el agente llega a la meta, recibe una **recompensa positiva grande (R)**.
- **Propagación de Valor:**
    - Cuando se aplica la fórmula, el Q(s, a) que llevó a la meta aumentará significativamente.
    - En episodios futuros, los estados y acciones que llevaron al next_state que tenía ese Q-value alto, verán cómo sus propios Q-values aumentan (gracias a $\gamma \cdot \max_{a'} Q(s',a')$).
    - Este proceso se propaga "hacia atrás" desde la meta. Es como si el valor positivo de la meta se fuera contagiando a los caminos que conducen a ella.
- **Ajuste de Expectativas:**
    - Si una ruta era "optimista" pero nunca llevó a recompensas, sus Q-values bajarán gradualmente.
    - Si una ruta consistentemente lleva a la meta, sus Q-values aumentarán y se estabilizarán, mostrando al agente el "mejor camino".

En Resumen:

El Q-learning con inicialización optimista permite al agente **explorar activamente** el entorno al principio. A través de la **fórmula de actualización iterativa**, el agente aprende los **valores verdaderos** de las acciones en cada estado, ajustando sus expectativas y finalmente convergiendo hacia una política óptima para navegar el laberinto y alcanzar sus objetivos.

Q-Learning con Selección de Acción UCB (Upper Confidence Bound)

Objetivo:

Entrenar a un agente que aprenda de manera eficiente en un entorno, balanceando de forma **inteligente** la **exploración** de acciones desconocidas y la **explotación** de las acciones que ya sabe que son buenas.

---

1. El Desafío: ¿Explorar o Explotar?

- **Exploración:** El agente prueba acciones nuevas para descubrir si hay caminos mejores o más recompensas.
- **Explotación:** El agente elige las acciones que ya sabe que le han dado buenas recompensas en el pasado.
- **UCB:** Es una estrategia que busca la **mejor de ambos mundos**. No solo elige la acción que parece mejor ahora, sino que también le da una "oportunidad" a las acciones que no ha probado mucho, ya que podrían ser mejores de lo que parece.

---

2. La Fórmula UCB: Confianza y Potencial

Para cada acción a en un estado s, el agente calcula un "valor UCB" y elige la acción con el valor más alto.

$$UCB(s,a)=Q(s,a)+c\cdot visit\_count(s,a)+\epsilon ln(total\_visits(s))$$

- **Q(s,a)**: El valor Q actual de tomar la acción a desde el estado s. Representa la **explotación** (lo que el agente ya sabe).
- **c**: Un coeficiente de exploración (c=2.0). Controla la **intensidad de la exploración**. Un valor más alto significa más exploración.
- **total_visits(s)**: El número total de veces que el agente ha visitado el estado s.
- **visit_count(s,a)**: El número de veces que el agente ha tomado la acción a desde el estado s.

- **visit_count(s,a)+ϵln(total_visits(s))** : El **término de exploración**.
    - Si una acción a ha sido probada **pocas veces** (visit_count(s, a) es bajo), este término será **grande**, dándole una bonificación a esa acción.
    - Si el estado s ha sido visitado **muchas veces** (total_visits(s) es alto), pero una acción a específica no, este término también se hace **más grande**, incentivando a probar esa acción "descuidada".
    - ϵ (1e-10): Un valor pequeño para evitar divisiones por cero.

---

3. El Código: Implementación en Python

El código mantiene contadores de visitas (visit_count y total_visits) para aplicar la fórmula UCB en cada paso de selección de acción:

Python

```
# 1. Al inicio de cada paso, si el estado es nuevo, elige acción aleatoria para inicializar.
if total_visits[state] == 0:
    action = np.random.randint(n_actions)
else:
    # 2. Si el estado ya ha sido visitado, calcula los valores UCB para todas las acciones
    ucb_values = q_table[state] + c * np.sqrt(
        np.log(total_visits[state]) / (visit_count[state] + 1e-10)
    )
    # 3. Elige la acción con el valor UCB más alto
    action = np.argmax(ucb_values)

# ... (Luego, el agente ejecuta la acción y actualiza Q-table como en Q-learning estándar)
# visit_count[state][action] += 1
# total_visits[state] += 1
# q_table[state][action] += alpha * (td_target - q_table[state][action])
```

Explicación Oral: ¿Cómo se Aplica al Entorno four-room-v0?

"Volvamos a nuestro laberinto de cuatro habitaciones. Con UCB, el agente no solo busca el camino más corto, sino que también se asegura de no ignorar posibles atajos que no ha explorado lo suficiente."

Escenario 1: Un Estado Nuevo o Pocas Veces Visto

- "El agente llega a una casilla (**estado s**) que acaba de descubrir o que solo ha visitado un par de veces."
- "Como total_visits[s] será bajo, pero el término de exploración de UCB seguirá siendo influyente, el agente tenderá a probar las **acciones que menos ha tomado** desde esa casilla."
- "**Impacto:** UCB fuerza una **exploración inicial efectiva**. El agente no se queda atascado en caminos conocidos si hay alternativas sin probar."

Escenario 2: Un Estado Frecuentemente Visitado, con Acciones Dispares

- "Imaginemos que el agente ha visitado una encrucijada (estado s) muchas veces. Conoce bien dos de los tres caminos, pero el tercero, el 'camino misterioso', solo lo ha probado una vez."
- "Aunque los Q(s, a) de los caminos conocidos son altos, el visit_count(s, a) del 'camino misterioso' es muy bajo. Esto hace que el término de exploración de UCB para ese camino sea **significativo**."
- "**Impacto:** UCB sigue animando al agente a volver a ese 'camino misterioso' de vez en cuando, incluso si el Q(s,a) de ese camino no parece tan prometedor al principio. Le da la oportunidad de demostrar su verdadero valor. Es una **exploración persistente**."

Escenario 3: Convergencia (Después de Mucho Entrenamiento)

- "Conforme el agente visita mucho un estado y prueba todas sus acciones un número suficiente de veces, los visit_count(s, a) de todas las acciones se vuelven grandes."
- "Esto hace que el **término de exploración se vuelva muy pequeño**, y la fórmula UCB se parezca más a simplemente elegir la acción con el Q(s,a) más alto."
- "**Impacto:** UCB transiciona automáticamente de la exploración a la **explotación**. El agente confía en lo que ha aprendido y se concentra en seguir los caminos óptimos."

¿Por qué UCB es Útil?

"UCB es una estrategia de exploración **más sofisticada** que simplemente la ϵ-greedy. Garantiza que todas las acciones en todos los estados sean exploradas lo suficiente para tener una estimación confiable de su valor, llevando a un **aprendizaje más eficiente y robusto**."

Q-Learning con Adaptación de Tasa de Aprendizaje por Gradiente

Objetivo:

Entrenar a un agente para que aprenda una política óptima, utilizando una estrategia de selección de acciones probabilística (Softmax) y ajustando la **velocidad de aprendizaje** basándose en qué tan "sorprendido" está el agente (magnitud del error).

---

1. Selección de Acción Suave: Softmax en Lugar de ϵ-greedy

- action = softmax_action(q_table[state], epsilon)
- **Enfoque:** En lugar de elegir una acción aleatoria con probabilidad ϵ o la mejor conocida, Softmax asigna una **probabilidad a cada acción** basándose en su Q-value.
- **¿Cómo funciona?** Las acciones con Q-values más altos tienen una **probabilidad mayor** de ser elegidas, pero las acciones con Q-values más bajos **no son ignoradas por completo**. Siempre hay una pequeña probabilidad de elegirlas.

- **epsilon (temperatura):** En Softmax, epsilon a menudo actúa como un parámetro de "temperatura".
    - Un epsilon **alto** (temperatura alta) hace que las probabilidades de todas las acciones sean más uniformes (más exploración).
    - Un epsilon **bajo** (temperatura baja) hace que las acciones con Q-values altos sean mucho más probables (más explotación).

---

2. Tasa de Aprendizaje Adaptativa: Aprender al Ritmo del Error

- **Concepto Central:** Cuando el agente comete un gran "error" (la realidad es muy diferente de lo que esperaba), debe aprender **rápido**. Cuando el error es pequeño, ya está cerca de la verdad, y solo necesita ajustar **ligeramente** su conocimiento.
- **El Error TD (td_error):** La diferencia entre la recompensa esperada (Q(s,a)) y la recompensa real más el valor descontado del siguiente estado. Es una medida de la "sorpresa" del agente. td_error=R+γ·a'maxQ(s',a')−Q(s,a)
- **Ajuste del alpha (effective_alpha):**
    - gradient_scale = np.tanh(abs(td_error))
    - effective_alpha = alpha * gradient_scale
    - La función **tangente hiperbólica (tanh)** "aplasta" el valor absoluto del td_error en un rango entre 0 y 1.
        - Si el td_error es **grande**, abs(td_error) es grande, y tanh(abs(td_error)) será cercano a **1**. Así, effective_alpha será casi igual a alpha (aprendizaje rápido).
        - Si el td_error es **pequeño**, abs(td_error) es pequeño, y tanh(abs(td_error)) será cercano a **0**. Así, effective_alpha será pequeña (aprendizaje lento).

---

3. El Código: Adaptando el Aprendizaje

Python

```
# 1. Selecciona acción usando Softmax
#    (asume una función softmax_action que devuelve la acción basada en Q-values y epsilon)
action = softmax_action(q_table[state], epsilon)

# ... (interacción con el entorno y cálculo de next_state, scalar_reward)

# 2. Calcula el Error de Diferencia Temporal (TD Error)
```

```
best_next = np.max(q_table[next_state])
td_error = scalar_reward + gamma * best_next - q_table[state][action]

# 3. Ajusta la tasa de aprendizaje (alpha) basada en la magnitud del TD error
gradient_scale = np.tanh(abs(td_error))
effective_alpha = alpha * gradient_scale

# 4. Aplica la actualización de Q-learning con el alpha efectivo
q_table[state][action] += effective_alpha * td_error
```

Explicación Oral: Aplicación en el Entorno four-room-v0

"Pensemos de nuevo en nuestro laberinto de cuatro habitaciones. Este agente no solo explora de forma más matizada con Softmax, sino que también es un 'estudiante' más inteligente: aprende intensamente cuando se equivoca mucho y solo refina cuando ya sabe bastante."

Escenario 1: Una Gran Sorpresa (Gran td_error)

- **Ejemplo:** El agente toma una acción en un estado (state, action) y, de repente, ¡encuentra la **meta** (gran scalar_reward) o choca brutalmente contra una pared (gran scalar_reward negativo)!
- **td_error:** La diferencia entre lo que esperaba y lo que realmente pasó es **enorme**.
- **effective_alpha:** tanh(abs(td_error)) será un valor cercano a **1**.
- **Impacto:** El q_table[state][action] se actualizará con una effective_alpha cercana a su alpha original (0.1). Esto significa que el agente **aprende muy rápidamente** de esa gran sorpresa, ajustando drásticamente su Q-value para esa acción en ese estado. "¡Wow, esto fue mucho mejor (o peor) de lo que pensaba, tengo que actualizar mi mapa inmediatamente!"

Escenario 2: Pequeña Corrección (Pequeño td_error)

- **Ejemplo:** El agente está siguiendo un camino que ya conoce bastante bien. Toma una acción, recibe una recompensa y llega a un siguiente estado que también es familiar.
- **td_error:** La diferencia entre lo que esperaba y lo que realmente pasó es **mínima**. Su predicción fue bastante precisa.
- **effective_alpha:** tanh(abs(td_error)) será un valor cercano a **0**.
- **Impacto:** El q_table[state][action] se actualizará con un effective_alpha muy pequeño. Esto significa que el agente **apenas modifica** su Q-value. "Mi mapa estaba casi bien, solo un pequeño ajuste."

¿Por qué es Útil esta Estrategia?

"Este enfoque ayuda a que el agente:

- **Aprenda más rápido** en las fases iniciales de exploración (cuando los errores son grandes).
- **Se estabilice mejor** en las fases posteriores, evitando oscilaciones en los valores Q una vez que ha aprendido la política, ya que las correcciones se vuelven más finas.
- La combinación de **Softmax** y alpha adaptativo crea un sistema de aprendizaje muy **dinámico y eficiente**."