

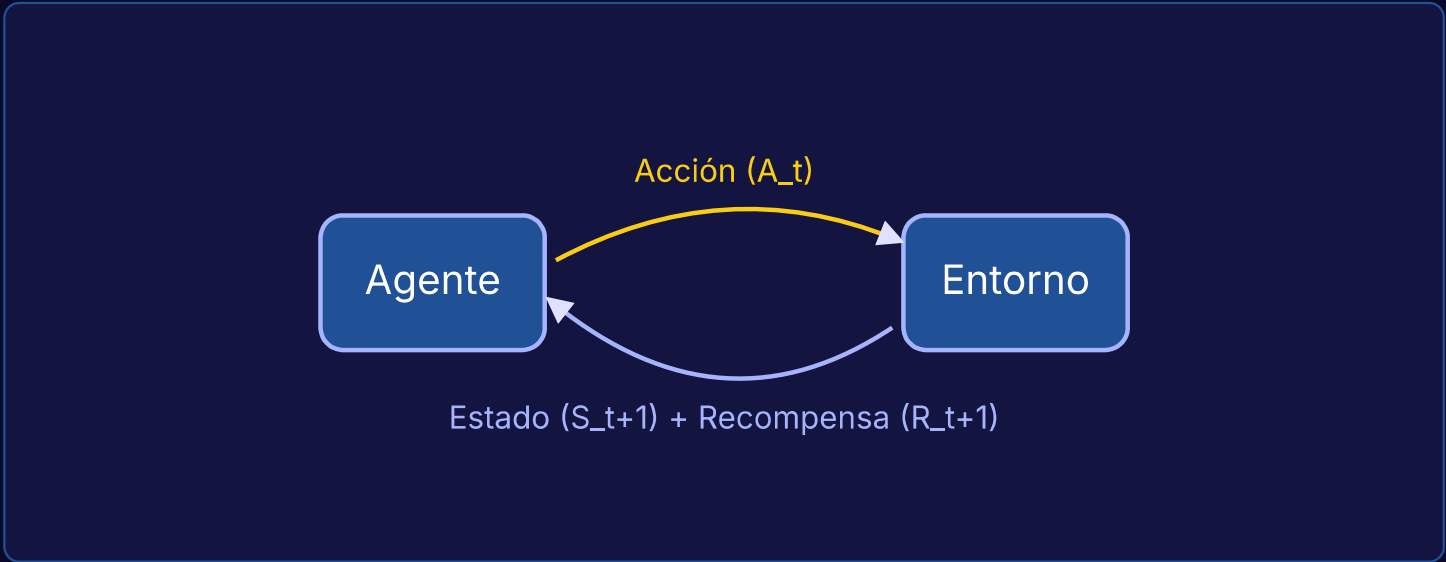
# Análisis de un Agente Autónomo en Laberintos mediante Aprendizaje por Refuerzo

Documentación técnica sobre la implementación de un agente inteligente para la resolución de problemas de navegación.

## Fundamentos del Aprendizaje por Refuerzo (RL)

El Aprendizaje por Refuerzo es un área del Machine Learning donde un agente computacional aprende a tomar decisiones a través de la interacción directa con un entorno. A diferencia de otros paradigmas, no se le proveen datos etiquetados.


El agente ejecuta acciones y recibe retroalimentación en forma de recompensas o penalizaciones. El objetivo del agente es desarrollar una política (una estrategia de toma de decisiones) que maximice la recompensa acumulada a lo largo del tiempo.



## Componentes Fundamentales del Modelo



### Agente

La entidad computacional que toma decisiones. En esta implementación, es la lógica que controla a los avatares .



### Entorno

El sistema con el que el agente interactúa. Aquí, es la estructura del laberinto, que define los estados posibles y las transiciones entre ellos.



### Acción

El conjunto de operaciones que el agente puede ejecutar. En este caso, un conjunto discreto de cuatro acciones: mover arriba, abajo, izquierda o derecha.



### Recompensa

Una señal escalar que el entorno envía al agente como evaluación de su última acción. Es la base para la optimización de la política del agente.

# Configuración de Parámetros del Modelo

## 🎯 Función de Recompensa



La función de recompensa define el objetivo del problema. Su diseño es crítico para guiar al agente hacia el comportamiento deseado.

- **GOAL\_REWARD = 100:** Una recompensa positiva alta al alcanzar el estado objetivo (la salida del laberinto).
- **WALL\_PENALTY = -20:** Una penalización negativa significativa por realizar una acción que resulta en una colisión con un muro.
- **MOVE\_REWARD = -0.01:** Una pequeña penalización por cada paso. Esto introduce un costo temporal, incentivando al agente a encontrar la ruta más corta.

## 🧠 Algoritmo Q-Learning



Se implementa el algoritmo Q-Learning, un método de RL sin modelo. Este busca aprender una función de valor acción-estado (Q-value) que estima la recompensa futura esperada al tomar una acción 'a' en un estado 's'.

- **LEARNING\_RATE = 0.1 (Tasa de Aprendizaje  $\alpha$ ):** Determina en qué medida la nueva información anula la información antigua. Un valor de 0.1 indica un aprendizaje gradual y estable.
- **DISCOUNT\_FACTOR = 0.95 (Factor de Descuento  $\gamma$ ):** Pondera la importancia de las recompensas futuras. Un valor cercano a 1 (0.95) indica que el agente tiene una visión a largo plazo.

## 🌀 Política de Exploración (Epsilon-Greedy)



Para asegurar que el agente explore suficientemente el espacio de estados, se utiliza una política  $\epsilon$ -greedy. Esta equilibra la explotación del conocimiento actual con la exploración de nuevas acciones.

- **EPSILON\_START = 1.0:** El valor inicial de  $\epsilon$  es 1.0, lo que significa que al principio, el 100% de las acciones son aleatorias (exploración pura).
- **EPSILON\_DECAY = 0.995:** Después de cada episodio,  $\epsilon$  se multiplica por este factor, reduciendo gradualmente la probabilidad de exploración.
- **EPSILON\_END = 0.01:** El valor mínimo de  $\epsilon$  es 0.01, asegurando que siempre haya una mínima probabilidad (1%) de exploración, incluso al final del entrenamiento.

# Lógica de Ejecución y Simulación

## Ciclo de Entrenamiento (Episodios)

El entrenamiento se realiza a lo largo de **1000 episodios** (EPISODES = 1000). Un episodio es un intento completo, desde el estado inicial hasta un estado terminal (meta o límite de pasos). Este número elevado de iteraciones es necesario para que la Q-Table converja a valores estables. Se visualizan los episodios **1, 50 y 200** como puntos de control para observar la evolución de la política del agente.

### Límite de Pasos por Episodio

Para prevenir bucles infinitos durante la fase de exploración, cada episodio está limitado por un número máximo de pasos. Este se calcula dinámicamente con la fórmula:

```
max_steps = width * height * 2
```

Para los laberintos de 19×9, el límite es de **342 pasos**. Si el agente alcanza este límite, el episodio termina y se considera no resuelto.

### Selección de Entornos (Laberintos)

El sistema implementa una lógica de selección de laberintos para garantizar la exposición a todos los entornos al inicio:

- 1. **Primeras 3 Partidas:** Se utiliza una cola pre-mezclada de los tres laberintos disponibles, asegurando que cada uno se juegue una vez sin repetición.
- 2. **Partidas Subsecuentes:** La selección del laberinto se realiza de forma uniformemente aleatoria entre los tres disponibles.

## Estructura del Código Fuente

### Librerías Utilizadas

- **Pygame:** Utilizada para la creación de la interfaz gráfica, la renderización del laberinto, el manejo de eventos de teclado y el control del tiempo.
- **NumPy:** Fundamental para la gestión eficiente de la Q-Table, que es una matriz numérica. Permite realizar operaciones matemáticas vectorizadas.
- **JSON:** Se emplea para la serialización y deserialización de los datos de estadísticas y logs, permitiendo guardarlos y cargarlos de forma estructurada.
- **OS, sys, time, datetime, random, subprocess:** Librerías estándar de Python para la interacción con el sistema operativo, manejo de tiempo, generación de números aleatorios y ejecución de procesos externos.

### Clase `Maze`

Abstrae el entorno del laberinto. Gestiona la representación del estado, las transiciones de estado y la función de recompensa. Su método principal, `step()`, procesa una acción del agente y devuelve el nuevo estado, la recompensa obtenida y una bandera booleana que indica si el estado es terminal.

```
def step(self, state, action):
    current_pos = self.get_pos_for_state(state)
    new_pos = self._get_new_pos_from_action(current_pos, action)

    if not (...condiciones de borde y pared...):
        return state, WALL_PENALTY, False, "Collision"

    if new_pos == self.goal_pos:
        return self.get_state_for_pos(new_pos), GOAL_REWARD, True, "Goal"

    return self.get_state_for_pos(new_pos), MOVE_REWARD, False, "Move"
```

## Clase `Agent`

Implementa la lógica del agente de RL. Su componente central es la **Q-Table**, una matriz que almacena el valor esperado de cada acción en cada estado del laberinto. Los métodos principales son:

- **choose\_action():** Implementa la política  **$\epsilon$ -greedy**. Con una probabilidad  $\epsilon$ , elige una acción al azar (exploración). Con probabilidad  $1-\epsilon$ , elige la mejor acción conocida según la Q-Table (explotación).
- **learn():** Aplica la regla de actualización de Q-Learning. Después de cada acción, ajusta el valor en la Q-Table para reflejar la recompensa obtenida y el valor máximo del nuevo estado, "refinando" así el conocimiento del agente.

```
def learn(self, state, action, reward, next_state):
    old_value = self.q_table[state, action]
    next_max = np.max(self.q_table[next_state, :])
    new_value = old_value + LEARNING_RATE * (reward + DISCOUNT_FACTOR * next_max - old_value)
    self.q_table[state, action] = new_value
```

## Clase `Game`

Actúa como el controlador principal de la simulación. Gestiona la inicialización de Pygame, el bucle principal del menú, la orquestación del ciclo de entrenamiento de 1000 episodios y la ejecución de la política aprendida. También invoca los módulos de registro de datos.

```
def run_game(self, maze_info, maze_name):
    ...
    agent = Agent(...)
    for episode in range(1, EPISODES + 1):
```

```
# Bucle de entrenamiento silencioso
while not done:
    action = agent.choose_action(state)
    next_state, reward, done, _ = maze.step(state, action)
    agent.learn(state, action, reward, next_state)
    ...
agent.decay_epsilon()
...
# Ejecución final con la política aprendida
final_run_data = self.run_single_episode(..., deterministic=True, ...)
```

## Generación de Resultados y Estadísticas

### Estructura del Archivo JSON

Al finalizar cada partida, se genera o actualiza un archivo `.json` que sirve como base de datos. A continuación se muestra una simulación de su estructura:

```
{
  "estadisticas_por_laberinto": { ... },
  "historial_partidas": [
    {
      "partida_numero": 1,
      "laberinto": "Ayuda_al_raton...",
      "resumen_resultado": { ... },
      "detalle_aprendizaje": {
        "intento_1": { // Log del primer intento... },
        "intento_50": { // Log del intento 50... }
      }
    }
  ]
}
```

### Generación de la Imagen de Salida

Tras la finalización del entrenamiento, se ejecuta una simulación final utilizando la política óptima aprendida (explotación pura). Al concluir esta simulación, se realiza una captura de la ventana de Pygame.

Esta imagen, guardada en formato `.png`, sirve como evidencia visual del rendimiento del agente, mostrando la trayectoria final que este determinó como la más eficiente.