Taller sobre clustering aglomerativo y DBSCAN

Agente Come Frutas

Integrantes: Ayala Ivonne, Cumbal Mateo, Garcés Boris, Morales David, Pereira Alicia

Introducción

El presente informe detalla el proceso de diseño, implementación y experimentación para el desarrollo de un agente autónomo de Inteligencia Artificial en el marco del proyecto "Agente Come-Frutas". El desafío se centra en un problema clásico de toma de decisiones secuenciales en un entorno dinámico y con riesgos, sirviendo como un caso de estudio práctico para la aplicación de técnicas avanzadas de Machine Learning.

El problema consiste en un entorno de rejilla de 5x5 en el que un agente debe aprender a navegar de manera eficiente. El objetivo principal es maximizar una puntuación recolectando "frutas" (que otorgan recompensas positivas) y evitando "venenos" (que imponen castigos negativos). La meta final es desarrollar una política de comportamiento óptima que permita al agente limpiar el tablero de todas las frutas, garantizando su supervivencia al esquivar todos los venenos presentes.

Para alcanzar este objetivo, el proyecto transitó por un riguroso proceso de experimentación, explorando múltiples paradigmas de la Inteligencia Artificial. Se inició con un enfoque en el Aprendizaje por Refuerzo (Reinforcement Learning), implementando y depurando algoritmos de vanguardia como Deep Q-Networks (DQN) y su variante mejorada, Double DQN (DDQN).

Frente a los desafíos clásicos de convergencia y estabilidad inherentes a RL, la investigación se expandió para incluir otras estrategias. Se exploraron los Algoritmos Genéticos, un enfoque basado en principios de evolución, y el Aprendizaje por Imitación, una potente técnica de aprendizaje supervisado que requirió el desarrollo de un "oráculo" experto basado en el algoritmo de búsqueda A*.

A continuación, se presenta el código documentado de la implementación final, reflejando la culminación de este profundo y multifacético proceso de desarrollo.

Q-Learning

```
# Importamos las bibliotecas necesarias
import pygame
import numpy as np
import random
import time
import os
```

```
# --- CONSTANTES DE CONFIGURACIÓN DEL JUEGO Y LA PANTALLA ---
# Estas constantes definen el tamaño del tablero, de cada celda y de la ventana del juego.
GRID_WIDTH = 5
GRID_HEIGHT = 5
CELL_SIZE = 120
SCREEN_WIDTH = GRID_WIDTH * CELL_SIZE
SCREEN_HEIGHT = GRID_HEIGHT * CELL_SIZE

# Colores (formato RGB) para los elementos de la interfaz.
COLOR_FONDO = (25, 25, 25)
COLOR_LINEAS = (40, 40, 40)
COLOR_AGENTE = (60, 100, 255)
COLOR_PARED = (80, 80, 80)
COLOR_TEXTO = (230, 230, 230)
COLOR_CURSOR = (255, 255, 0)

# --- PARÁMETROS DEL APRENDIZAJE POR REFUERZO (Q-LEARNING) ---
# Estos son los "hiperparámetros" que controlan cómo aprende el agente.
RECOMPENSA_FRUTA = 100         # Puntuación alta por encontrar una fruta.
CASTIGO_VENENO = -100           # Castigo fuerte por tocar un veneno.
RECOMPENSA_MOVIMIENTO = -0.1    # Pequeño castigo por cada movimiento para incentivar la efi
ALPHA = 0.1                     # Tasa de aprendizaje.
GAMMA = 0.9                     # Factor de descuento.
EPSILON = 1.0                   # Tasa de exploración inicial.
EPSILON_DECAY = 0.9995          # Factor de decaimiento de epsilon.
MIN_EPSILON = 0.01              # Mínima tasa de exploración.
NUM_EPISODIOS_ENTRENAMIENTO = 20000 # Número de partidas que jugará el agente para aprender.
```

```python
# --- CLASE DEL AGENTE ---
# Define el "cerebro" del agente. Contiene la Tabla Q y la lógica para aprender y decidir.
class AgenteQLearning:
    def __init__(self, num_estados, num_acciones):
        self.num_acciones = num_acciones
        # La Tabla Q es una matriz que almacena el "valor" de cada acción en cada estado posi
        # Aquí, el estado está definido por la posición (x, y) del agente en el tablero.
        self.q_table = np.zeros((num_estados[0], num_estados[1], num_acciones))
        self.epsilon = EPSILON  # Tasa de exploración (curiosidad).

    def elegir_accion(self, estado):
        """Decide qué acción tomar usando la estrategia epsilon-greedy."""
        # Con probabilidad epsilon, toma una acción aleatoria (exploración).
        if random.uniform(0, 1) < self.epsilon:
            return random.randint(0, self.num_acciones - 1)
        # De lo contrario, elige la mejor acción conocida según la Tabla Q (explotación).
        else:
            return np.argmax(self.q_table[estado])

    def actualizar_q_table(self, estado, accion, recompensa, nuevo_estado):
        """Actualiza el valor en la Tabla Q usando la fórmula de Bellman."""
        valor_antiguo = self.q_table[estado][accion]
        # El valor futuro es el máximo valor Q que se puede obtener desde el nuevo estado.
        valor_futuro_maximo = np.max(self.q_table[nuevo_estado])

        # Fórmula de Q-Learning: se actualiza el valor antiguo basado en la recompensa
        # obtenida y el valor futuro esperado.
        nuevo_q = valor_antiguo + ALPHA * (
            recompensa + GAMMA * valor_futuro_maximo - valor_antiguo
        )
        self.q_table[estado][accion] = nuevo_q

    def decaimiento_epsilon(self):
        """Reduce gradualmente el valor de epsilon para pasar de explorar a explotar."""
        if self.epsilon > MIN_EPSILON:
            self.epsilon *= EPSILON_DECAY
```

```python
# --- CLASE DEL ENTORNO ---
# Define las reglas del juego, el tablero y cómo interactúa el agente con él.
class EntornoGrid:
    def __init__(self):
        self.agente_pos = (0, 0)
```

```
        # Usamos 'sets' para un manejo eficiente de las posiciones de los objetos.
        self.frutas = set()
        self.venenos = set()
        self.paredes = set()
        self.reset_a_configuracion_inicial()

    def reset_a_configuracion_inicial(self):
        """Resetea la posición del agente al inicio (esquina superior izquierda)."""
        self.agente_pos = (0, 0)
        return self.agente_pos

    def limpiar_entorno(self):
        """Elimina todos los objetos del tablero."""
        self.frutas.clear()
        self.venenos.clear()
        self.paredes.clear()

    def step(self, accion, modo_juego):
        """
        Ejecuta un paso en el juego.
        El agente toma una 'accion' y el entorno devuelve el 'nuevo_estado',
        la 'recompensa' y si el juego ha 'terminado'.
        """
        x, y = self.agente_pos
        # Acciones: 0=arriba, 1=abajo, 2=izquierda, 3=derecha
        if accion == 0: y -= 1
        elif accion == 1: y += 1
        elif accion == 2: x -= 1
        elif accion == 3: x += 1

        # Comprueba si el movimiento es válido (dentro de los límites y no choca con una pare
        if (x < 0 or x >= GRID_WIDTH or y < 0 or y >= GRID_HEIGHT or (x, y) in self.paredes)
            # Si el movimiento es inválido, el agente no se mueve y recibe una pequeña penali
            return self.agente_pos, RECOMPENSA_MOVIMIENTO, False

        # Actualiza la posición del agente si el movimiento es válido.
        self.agente_pos = (x, y)
        nuevo_estado = self.agente_pos
        terminado = False

        if nuevo_estado in self.frutas:
            recompensa = RECOMPENSA_FRUTA
```

```
            self.frutas.remove(nuevo_estado)
            # El episodio solo termina con éxito si ya no quedan más frutas.
            if not self.frutas:
                terminado = True
        elif nuevo_estado in self.venenos:
            recompensa = CASTIGO_VENENO
            terminado = True  # Tocar un veneno siempre termina el juego.
        else:
            recompensa = RECOMPENSA_MOVIMIENTO

        return nuevo_estado, recompensa, terminado

    def dibujar(self, pantalla, modo_juego, cursor_pos, img_fruta, img_veneno, img_pared, img
        """Dibuja todos los elementos del juego en la pantalla."""
        pantalla.fill(COLOR_FONDO)

        # Dibuja la cuadrícula.
        for x in range(0, SCREEN_WIDTH, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (x, 0), (x, SCREEN_HEIGHT))
        for y in range(0, SCREEN_HEIGHT, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (0, y), (SCREEN_WIDTH, y))

        # Dibuja las paredes, frutas, venenos y el agente usando sus imágenes.
        for pared in self.paredes:
            pantalla.blit(img_pared, (pared[0] * CELL_SIZE, pared[1] * CELL_SIZE))
        for fruta in self.frutas:
            pantalla.blit(img_fruta, (fruta[0] * CELL_SIZE, fruta[1] * CELL_SIZE))
        for veneno in self.venenos:
            pantalla.blit(img_veneno, (veneno[0] * CELL_SIZE, veneno[1] * CELL_SIZE))
        pantalla.blit(img_agente, (self.agente_pos[0] * CELL_SIZE, self.agente_pos[1] * CELL_

        # En modo SETUP, dibuja un cursor para indicar dónde se colocarán los objetos.
        if modo_juego == "SETUP":
            cursor_rect = pygame.Rect(
                cursor_pos[0] * CELL_SIZE, cursor_pos[1] * CELL_SIZE, CELL_SIZE, CELL_SIZE
            )
            pygame.draw.rect(pantalla, COLOR_CURSOR, cursor_rect, 3)

        # Dibuja la información de ayuda y el modo de juego actual en la parte inferior.
        font = pygame.font.Font(None, 24)
        texto_modo = font.render(f"Modo: {modo_juego}", True, COLOR_TEXTO)
        controles_setup = font.render(
```

```
            "SETUP: Mover con flechas. F=Fruta, V=Veneno, W=Pared. 'C' para limpiar.", True,
        )
        controles_run = font.render(
            "'T' para Entrenar, 'P' para Jugar, 'S' para Setup.", True, COLOR_TEXTO
        )

        pantalla.blit(texto_modo, (10, SCREEN_HEIGHT + 5))
        pantalla.blit(controles_setup, (10, SCREEN_HEIGHT + 30))
        pantalla.blit(controles_run, (10, SCREEN_HEIGHT + 55))
```

```
# --- FUNCIÓN PRINCIPAL DEL JUEGO ---
# Orquesta todo el juego: inicialización, bucle principal, manejo de eventos y modos.
def main():
    pygame.init()
    pantalla = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
    pygame.display.set_caption("Agente Come-Frutas  vs  (Q-Learning)")

    # --- Carga de imágenes ---
    # Intenta cargar los archivos de imagen. Si no los encuentra, usa cuadrados de colores co
    try:
        ruta_fruta = os.path.join(os.path.dirname(__file__), "fruta.png")
        img_fruta_original = pygame.image.load(ruta_fruta).convert_alpha()
        img_fruta = pygame.transform.scale(img_fruta_original, (CELL_SIZE, CELL_SIZE))
    except pygame.error:
        print("Advertencia: No se encontró 'fruta.png'. Se usará un cuadrado verde.")
        img_fruta = pygame.Surface((CELL_SIZE, CELL_SIZE)); img_fruta.fill((40, 200, 40))

    try:
        ruta_veneno = os.path.join(os.path.dirname(__file__), "veneno.png")
        img_veneno_original = pygame.image.load(ruta_veneno).convert_alpha()
        img_veneno = pygame.transform.scale(img_veneno_original, (CELL_SIZE, CELL_SIZE))
    except pygame.error:
        print("Advertencia: No se encontró 'veneno.png'. Se usará un cuadrado rojo.")
        img_veneno = pygame.Surface((CELL_SIZE, CELL_SIZE)); img_veneno.fill((255, 50, 50))

    try:
        ruta_pared = os.path.join(os.path.dirname(__file__), "pared.png")
        img_pared_original = pygame.image.load(ruta_pared).convert_alpha()
        img_pared = pygame.transform.scale(img_pared_original, (CELL_SIZE, CELL_SIZE))
    except pygame.error:
        print("Advertencia: No se encontró 'pared.png'. Se usará un cuadrado gris.")
```

```
        img_pared = pygame.Surface((CELL_SIZE, CELL_SIZE)); img_pared.fill(COLOR_PARED)

    try:
        ruta_agente = os.path.join(os.path.dirname(__file__), "agente.png")
        img_agente_original = pygame.image.load(ruta_agente).convert_alpha()
        img_agente = pygame.transform.scale(img_agente_original, (CELL_SIZE, CELL_SIZE))
    except pygame.error:
        print("Advertencia: No se encontró 'agente.png'. Se usará un cuadrado azul.")
        img_agente = pygame.Surface((CELL_SIZE, CELL_SIZE)); img_agente.fill(COLOR_AGENTE)

    # Inicialización del entorno y el agente.
    entorno = EntornoGrid()
    agente = AgenteQLearning(num_estados=(GRID_HEIGHT, GRID_WIDTH), num_acciones=4)

    reloj = pygame.time.Clock()
    corriendo = True
    modo_juego = "SETUP"  # El juego comienza en modo de configuración.
    cursor_pos = [0, 0]

    # Guarda la configuración inicial del tablero para poder resetearlo.
    frutas_iniciales = entorno.frutas.copy()
    venenos_iniciales = entorno.venenos.copy()

    # --- BUCLE PRINCIPAL DEL JUEGO ---
    while corriendo:
        for evento in pygame.event.get():
            if evento.type == pygame.QUIT:
                corriendo = False

            # Manejo de eventos de teclado para cambiar de modo y configurar el tablero.
            if evento.type == pygame.KEYDOWN:
                # --- MODO ENTRENAMIENTO (T) ---
                if evento.key == pygame.K_t:
                    if modo_juego != "TRAINING":
                        print("--- INICIANDO ENTRENAMIENTO ---")
                        modo_juego = "TRAINING"
                        # Crea un nuevo agente con una Tabla Q vacía.
                        agente = AgenteQLearning(num_estados=(GRID_HEIGHT, GRID_WIDTH), num_a
                        # Muestra un mensaje de "Entrenando..." en pantalla.
                        pantalla.fill(COLOR_FONDO)
                        font = pygame.font.Font(None, 50)
                        texto_entrenando = font.render("Entrenando...", True, COLOR_TEXTO)
```

```
                    rect = texto_entrenando.get_rect(center=(SCREEN_WIDTH / 2, SCREEN_HE
                    pantalla.blit(texto_entrenando, rect)
                    pygame.display.flip()

                    # Bucle de entrenamiento principal.
                    for episodio in range(NUM_EPISODIOS_ENTRENAMIENTO):
                        # Resetea el entorno a la configuración definida en modo SETUP.
                        entorno.frutas = frutas_iniciales.copy()
                        entorno.venenos = venenos_iniciales.copy()
                        estado = entorno.reset_a_configuracion_inicial()
                        terminado = False

                        # Bucle de una partida (episodio).
                        while not terminado:
                            accion = agente.elegir_accion(estado)
                            nuevo_estado, recompensa, terminado = entorno.step(accion, "
                            agente.actualizar_q_table(estado, accion, recompensa, nuevo_
                            estado = nuevo_estado

                        # Reduce epsilon al final de cada episodio.
                        agente.decaimiento_epsilon()

                        # Imprime el progreso cada 1000 episodios.
                        if (episodio + 1) % 1000 == 0:
                            print(f"Episodio: {episodio + 1}/{NUM_EPISODIOS_ENTRENAMIENTO

                    print("--- ENTRENAMIENTO COMPLETADO ---")
                    # Prepara el tablero para la demostración del agente ya entrenado.
                    entorno.frutas = frutas_iniciales.copy()
                    entorno.venenos = venenos_iniciales.copy()
                    entorno.reset_a_configuracion_inicial()
                    agente.epsilon = 0  # Modo experto: solo explotación, sin acciones al
                    modo_juego = "PLAYING"

                # --- MODO JUEGO (P) ---
                elif evento.key == pygame.K_p:
                    print("--- MODO JUEGO (AGENTE ENTRENADO) ---")
                    modo_juego = "PLAYING"
                    # Resetea el tablero a la configuración inicial.
                    entorno.frutas = frutas_iniciales.copy()
                    entorno.venenos = venenos_iniciales.copy()
                    entorno.reset_a_configuracion_inicial()
```

```
                    agente.epsilon = 0 # El agente usará su conocimiento sin explorar.

                # --- MODO SETUP (S) ---
                elif evento.key == pygame.K_s:
                    print("--- MODO SETUP ---")
                    modo_juego = "SETUP"

                # Lógica para configurar el tablero en modo SETUP.
                if modo_juego == "SETUP":
                    if evento.key == pygame.K_UP: cursor_pos[1] = max(0, cursor_pos[1] - 1)
                    elif evento.key == pygame.K_DOWN: cursor_pos[1] = min(GRID_HEIGHT - 1, cu
                    elif evento.key == pygame.K_LEFT: cursor_pos[0] = max(0, cursor_pos[0] -
                    elif evento.key == pygame.K_RIGHT: cursor_pos[0] = min(GRID_WIDTH - 1, cu

                    pos_celda = tuple(cursor_pos)
                    # Tecla F: Añade o quita una fruta.
                    if evento.key == pygame.K_f:
                        if pos_celda in entorno.frutas: entorno.frutas.remove(pos_celda)
                        else:
                            entorno.frutas.add(pos_celda)
                            entorno.venenos.discard(pos_celda); entorno.paredes.discard(pos_
                    # Tecla V: Añade o quita un veneno.
                    elif evento.key == pygame.K_v:
                        if pos_celda in entorno.venenos: entorno.venenos.remove(pos_celda)
                        else:
                            entorno.venenos.add(pos_celda)
                            entorno.frutas.discard(pos_celda); entorno.paredes.discard(pos_ce
                    # Tecla W: Añade o quita una pared.
                    elif evento.key == pygame.K_w:
                        if pos_celda in entorno.paredes: entorno.paredes.remove(pos_celda)
                        else:
                            entorno.paredes.add(pos_celda)
                            entorno.frutas.discard(pos_celda); entorno.venenos.discard(pos_ce
                    # Tecla C: Limpia el tablero.
                    elif evento.key == pygame.K_c:
                        print("--- TABLERO LIMPIO ---")
                        entorno.limpiar_entorno()

                    # Actualiza la configuración inicial guardada cada vez que se hace un car
                    frutas_iniciales = entorno.frutas.copy()
                    venenos_iniciales = entorno.venenos.copy()
```

```
        # --- Lógica de juego que se ejecuta en cada frame ---
        # Lógica del juego en modo PLAYING: el agente toma decisiones.
        if modo_juego == "PLAYING":
            if entorno.frutas:
                estado = entorno.agente_pos
                accion = agente.elegir_accion(estado)
                _, _, terminado = entorno.step(accion, "PLAYING")
                if terminado:
                    if not entorno.frutas:
                        print("¡Todas las frutas recolectadas! Volviendo a modo SETUP.")
                    else:
                        print("Juego terminado (veneno). Volviendo a modo SETUP.")
                    modo_juego = "SETUP"
                time.sleep(0.1) # Pequeña pausa para ver el movimiento del agente.
            else:
                modo_juego = "SETUP"
                print("No hay frutas en el tablero. Volviendo a modo SETUP.")

        # Dibuja la pantalla en cada frame, excepto durante el entrenamiento.
        if modo_juego != "TRAINING":
            entorno.dibujar(pantalla, modo_juego, tuple(cursor_pos), img_fruta, img_veneno, i
            pygame.display.flip()

        reloj.tick(60) # Limita el juego a 60 frames por segundo.

    pygame.quit()
```

```
# Punto de entrada del programa.
if __name__ == "__main__":
    main()
```

DQN

agent.py

```
# agent.py
"""
Implementación del agente DQN (Deep Q-Network) con arquitectura CNN.

Este módulo contiene la implementación completa del algoritmo DQN, incluyendo:
- Red neuronal convolucional para procesamiento de estados espaciales.
```

```
- Sistema de memoria de replay para entrenamiento estable.
- Estrategia epsilon-greedy para balancear exploración/explotación.
- Red objetivo para estabilizar el cálculo de los valores Q.
- Optimización con el optimizador Adam.

El agente está diseñado específicamente para problemas de navegación en grillas
donde el estado se representa como imágenes multi-canal, aprovechando las
capacidades de las CNNs para reconocer patrones espaciales.

Características principales:
- Arquitectura CNN optimizada para grillas pequeñas.
- Memoria de replay para descorrelacionar experiencias.
- Actualización periódica de la red objetivo.
- Técnicas de estabilización (gradient clipping, target network).
- Sistema de guardado/carga de modelos entrenados.

Referencias:
- DQN: Mnih et al. (2015) "Human-level control through deep reinforcement learning"
"""

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
from collections import deque

# --- 1. RED NEURONAL CONVOLUCIONAL PARA DQN ---
class CNN_DQN(nn.Module):
    """
    Red neuronal convolucional optimizada para Q-learning en entornos de grilla.

    Esta arquitectura está diseñada para procesar estados representados como
    tensores 3D (canales x altura x anchura).

    Arquitectura:
    1. **Capas Convolucionales**: Para extraer características espaciales.
       - Conv1: 3->16 canales.
       - Conv2: 16->32 canales.

    2. **Capas Completamente Conectadas**: Para tomar decisiones basadas en las caracterísitic
       - FC1: 256 neuronas.
```

```
    - FC2: Salida de valores Q para cada acción.

Args:
    h (int): Altura de la grilla de entrada.
    w (int): Anchura de la grilla de entrada.
    outputs (int): Número de acciones posibles.
"""

def __init__(self, h, w, outputs):
    super(CNN_DQN, self).__init__()

    # --- CAPAS CONVOLUCIONALES ---
    self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
    self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)

    # --- CÁLCULO DINÁMICO DEL TAMAÑO DE CARACTERÍSTICAS ---
    def conv2d_size_out(size, kernel_size=3, stride=1, padding=1):
        return (size + 2 * padding - kernel_size) // stride + 1

    convw = conv2d_size_out(conv2d_size_out(w))
    convh = conv2d_size_out(conv2d_size_out(h))
    linear_input_size = convw * convh * 32

    # --- CAPAS COMPLETAMENTE CONECTADAS ---
    self.fc1 = nn.Linear(linear_input_size, 256)
    self.fc2 = nn.Linear(256, outputs)

def forward(self, x):
    """
    Propagación hacia adelante de la red.

    Procesa el estado de entrada a través de las capas para generar
    valores Q para cada acción posible.

    Args:
        x (torch.Tensor): Estado de entrada con forma (batch, 3, height, width).

    Returns:
        torch.Tensor: Valores Q para cada acción con forma (batch, num_actions).
    """
    x = nn.functional.relu(self.conv1(x))
    x = nn.functional.relu(self.conv2(x))
```

```
        x = x.view(x.size(0), -1)
        x = nn.functional.relu(self.fc1(x))
        return self.fc2(x)


# --- 2. AGENTE DQN CON MEMORIA DE REPLAY Y RED OBJETIVO ---
class Agent:
    """
    Agente de aprendizaje por refuerzo que implementa el algoritmo DQN.

    Este agente combina varias técnicas clave de deep reinforcement learning:

    **Componentes principales:**
    1. **Red Principal**: Se entrena activamente y decide las acciones.
    2. **Red Objetivo**: Una copia de la red principal que se actualiza lentamente,
       proporcionando targets estables para el entrenamiento y reduciendo oscilaciones.
    3. **Memoria de Replay**: Almacena experiencias para un aprendizaje más estable.
    4. **Estrategia Epsilon-Greedy**: Balancea entre explorar el entorno y explotar el conoci

    Args:
        state_shape (tuple): Forma del estado (canales, altura, anchura).
        action_size (int): Número de acciones posibles en el entorno.
    """

    def __init__(self, state_shape, action_size):
        # --- CONFIGURACIÓN BÁSICA ---
        self.state_shape = state_shape
        self.action_size = action_size

        # --- MEMORIA DE REPLAY ---
        # Almacena tuplas de (estado, acción, recompensa, siguiente_estado, terminado).
        self.memory = deque(maxlen=20000)

        # --- HIPERPARÁMETROS DE APRENDIZAJE ---
        self.gamma = 0.99
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.9995
        self.learning_rate = 0.0001
        self.update_target_every = 5

        # --- INICIALIZACIÓN DE REDES NEURONALES ---
```

```
        h, w = state_shape[1], state_shape[2]
        self.model = CNN_DQN(h, w, action_size)
        self.target_model = CNN_DQN(h, w, action_size)
        self.update_target_network()

        # --- CONFIGURACIÓN DE OPTIMIZACIÓN ---
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.learning_rate)
        self.criterion = nn.MSELoss()
        self.steps_done = 0

    def update_target_network(self):
        """
        Actualiza la red objetivo copiando los pesos de la red principal.

        Esta operación es fundamental en DQN para mantener los targets estables
        durante el entrenamiento, evitando que el objetivo cambie en cada paso.
        """
        self.target_model.load_state_dict(self.model.state_dict())

    def remember(self, state, action, reward, next_state, done):
        """
        Almacena una experiencia en la memoria de replay.

        Esto permite al agente aprender de un conjunto de experiencias pasadas y
        no correlacionadas, lo que estabiliza el entrenamiento.

        Args:
            state (np.array): Estado actual.
            action (int): Acción tomada.
            reward (float): Recompensa recibida.
            next_state (np.array): Estado resultante.
            done (bool): True si el episodio terminó.
        """
        self.memory.append((state, action, reward, next_state, done))

    def choose_action(self, state, explore=True):
        """
        Selecciona una acción usando la estrategia epsilon-greedy.

        - **Exploración**: Con probabilidad epsilon, elige una acción al azar.
        - **Explotación**: Con probabilidad 1-epsilon, elige la mejor acción según la red.
```

```
        Args:
            state (np.array): Estado actual del entorno.
            explore (bool): Permite la exploración. Poner en False para la demostración.

        Returns:
            int: La acción seleccionada.
        """
        self.steps_done += 1

        if explore and np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)

        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        with torch.no_grad():
            action_values = self.model(state_tensor)

        return np.argmax(action_values.cpu().data.numpy())

    def replay(self, batch_size):
        """
        Entrena la red neuronal usando un lote de experiencias de la memoria.

        Este es el núcleo del algoritmo de aprendizaje DQN.

        Proceso:
        1. Muestrear un lote (batch) aleatorio de experiencias.
        2. Calcular los valores Q actuales (predicciones) con la red principal.
        3. Calcular los valores Q objetivo (targets) usando la red objetivo.
        4. Optimizar la red principal para minimizar la diferencia entre predicciones y targe

        Args:
            batch_size (int): Número de experiencias a usar para el entrenamiento.
        """
        if len(self.memory) < batch_size:
            return

        minibatch = random.sample(self.memory, batch_size)

        states = torch.FloatTensor(np.array([e[0] for e in minibatch]))
        actions = torch.LongTensor([e[1] for e in minibatch]).unsqueeze(1)
        rewards = torch.FloatTensor([e[2] for e in minibatch]).unsqueeze(1)
        next_states = torch.FloatTensor(np.array([e[3] for e in minibatch]))
```

```
        dones = torch.BoolTensor([e[4] for e in minibatch]).unsqueeze(1)

        current_q_values = self.model(states).gather(1, actions)

        # --- CÁLCULO DEL TARGET SEGÚN DQN ---
        # La red objetivo calcula el valor máximo del siguiente estado.
        with torch.no_grad():
            next_q_values = self.target_model(next_states).max(1)[0].unsqueeze(1)

        # Ecuación de Bellman para el target: R + gamma * max_Q(s', a')
        target_q_values = rewards + (self.gamma * next_q_values * (~dones))

        loss = self.criterion(current_q_values, target_q_values)

        self.optimizer.zero_grad()
        loss.backward()

        # Gradient clipping para prevenir gradientes explosivos y estabilizar.
        torch.nn.utils.clip_grad_value_(self.model.parameters(), 1)
        self.optimizer.step()

        # Decaimiento de epsilon para reducir la exploración.
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def load(self, name):
        """
        Carga los pesos de un modelo entrenado desde un archivo.

        Args:
            name (str): Ruta al archivo de pesos del modelo (.pth).
        """
        self.model.load_state_dict(torch.load(name))
        self.update_target_network()

    def save(self, name):
        """
        Guarda los pesos del modelo actual en un archivo.

        Args:
            name (str): Ruta donde se guardará el archivo de pesos (.pth).
        """
```

```
        torch.save(self.model.state_dict(), name)
```

dqn_agente_comefrutas.py

```
# dqn_agente_comefrutas.py
"""
Interfaz gráfica de demostración para agente DQN (Deep Q-Network).

Este módulo proporciona una interfaz visual interactiva para demostrar el
comportamiento de un agente DQN entrenado en el problema de recolección de frutas.
A diferencia del DDQN, esta implementación utiliza DQN clásico con una sola red.

Características principales:
- Interfaz de configuración interactiva para crear escenarios personalizados
- Visualización en tiempo real del comportamiento del agente entrenado
- Sistema de dos modos: configuración (SETUP) y ejecución (PLAYING)
- Compatibilidad con modelos DQN preentrenados
- Interfaz de usuario intuitiva con controles de teclado y mouse

El sistema está diseñado para:
- Demostraciones educativas del comportamiento de IA
- Validación visual del rendimiento del agente
- Experimentación rápida con diferentes configuraciones de entorno
- Evaluación cualitativa de estrategias aprendidas

Diferencias con DDQN:
- Utiliza una sola red neuronal (no red objetivo separada)
- Implementación más simple del algoritmo Q-learning
- Compatible con modelos entrenados usando DQN clásico
"""

import pygame
import numpy as np
import os
import time
from agent import Agent

# --- CONFIGURACIÓN DE LA INTERFAZ VISUAL ---
"""Parámetros de configuración para la ventana y visualización."""
GRID_WIDTH = 5          # Ancho de la grilla en número de celdas
GRID_HEIGHT = 5         # Alto de la grilla en número de celdas
```

```
CELL_SIZE = 120         # Tamaño de cada celda en píxeles (120x120)
SCREEN_WIDTH = GRID_WIDTH * CELL_SIZE    # Ancho total de la ventana (600px)
SCREEN_HEIGHT = GRID_HEIGHT * CELL_SIZE  # Alto total de la ventana (600px)

# --- ESQUEMA DE COLORES ---
"""Paleta de colores para una interfaz moderna y legible."""
COLOR_FONDO = (25, 25, 25)        # Fondo oscuro para reducir fatiga visual
COLOR_LINEAS = (40, 40, 40)       # Líneas de grilla sutiles
COLOR_CURSOR = (255, 255, 0)      # Cursor amarillo brillante para visibilidad
COLOR_TEXTO = (230, 230, 230)     # Texto claro sobre fondo oscuro


class EntornoGrid:
    """
    Entorno de grilla especializado para demostración de agentes DQN.

    Esta clase maneja tanto la lógica del entorno como su representación visual,
    proporcionando una plataforma completa para demostrar el comportamiento de
    agentes DQN entrenados en problemas de navegación y recolección.

    Características del entorno:
    - Grilla bidimensional con elementos configurables
    - Gestión de colisiones y límites
    - Sistema de recompensas integrado
    - Representación visual con Pygame
    - Compatibilidad con formato de estado DQN (tensor 3D)

    Elementos del entorno:
    - Agente: Entidad controlada por IA que debe recolectar frutas
    - Frutas: Objetivos que otorgan recompensas positivas
    - Venenos: Obstáculos que causan penalizaciones y reseteo
    - Paredes: Barreras físicas que bloquean el movimiento

    Attributes:
        size (int): Tamaño de la grilla (siempre cuadrada)
        agent_pos (tuple): Posición actual del agente (fila, columna)
        frutas (set): Conjunto de posiciones que contienen frutas
        venenos (set): Conjunto de posiciones que contienen venenos
        paredes (set): Conjunto de posiciones que contienen paredes
    """

    def __init__(self):
```

```
        """
        Inicializa el entorno con configuración por defecto.

        El entorno comienza con:
        - Agente en posición (0,0) - esquina superior izquierda
        - Todos los conjuntos de elementos vacíos
        - Tamaño de grilla determinado por GRID_WIDTH
        """
        self.size = GRID_WIDTH
        self.agent_pos = (0, 0)  # Posición inicial estándar
        self.frutas = set()      # Conjunto vacío inicialmente
        self.venenos = set()     # Conjunto vacío inicialmente
        self.paredes = set()     # Conjunto vacío inicialmente

    def reset_a_configuracion_inicial(self):
        """
        Reinicia solo la posición del agente sin modificar el entorno.

        Esta función es útil para comenzar nuevos episodios manteniendo
        la misma configuración de elementos (frutas, venenos, paredes)
        establecida durante el modo setup.

        Returns:
            np.array: Estado inicial del entorno después del reset
        """
        self.agent_pos = (0, 0)
        return self.get_state()

    def limpiar_entorno(self):
        """
        Elimina todos los elementos del entorno excepto el agente.

        Función de utilidad para limpiar completamente el entorno
        y comenzar una nueva configuración desde cero. Útil en
        modo setup para crear nuevos escenarios rápidamente.
        """
        self.frutas.clear()
        self.venenos.clear()
        self.paredes.clear()

    def step(self, accion):
        """
```

```
        Ejecuta una acción del agente y actualiza el estado del entorno.

        Este método implementa la lógica principal del entorno, incluyendo:
        - Procesamiento de movimientos del agente
        - Detección de colisiones con paredes y límites
        - Cálculo de recompensas según las interacciones
        - Gestión de condiciones de terminación
        - Manejo especial de venenos (penalización + reset)

        Args:
            accion (int): Acción a ejecutar por el agente
                0: Mover hacia arriba (fila-1)
                1: Mover hacia abajo (fila+1)
                2: Mover hacia la izquierda (columna-1)
                3: Mover hacia la derecha (columna+1)

        Returns:
            tuple: (nuevo_estado, recompensa, episodio_terminado)
                - nuevo_estado (np.array): Estado resultante
                - recompensa (float): Recompensa obtenida
                - episodio_terminado (bool): True si completó o falló

        Sistema de recompensas:
            - Colisión con pared/límite: -0.1 (movimiento inválido)
            - Movimiento válido: -0.05 (costo de vida)
            - Tocar veneno: -10.0 (penalización severa + reset a origen)
            - Recolectar fruta: +1.0 (recompensa por objetivo)
            - Completar nivel: +10.0 adicional (todas las frutas recolectadas)
        """
        # Obtener posición actual del agente
        fila, col = self.agent_pos

        # Calcular nueva posición según la acción
        if accion == 0:      # Arriba
            fila -= 1
        elif accion == 1:    # Abajo
            fila += 1
        elif accion == 2:    # Izquierda
            col -= 1
        elif accion == 3:    # Derecha
            col += 1
```

```
        # Verificar colisiones con límites de grilla o paredes
        if (
            fila < 0                        # Límite superior
            or fila >= GRID_HEIGHT          # Límite inferior
            or col < 0                      # Límite izquierdo
            or col >= GRID_WIDTH            # Límite derecho
            or (fila, col) in self.paredes  # Colisión con pared
        ):
            # Movimiento inválido: penalización menor, mantener posición
            return self.get_state(), -0.1, False

        # Movimiento válido: actualizar posición
        x, y = fila, col
        self.agent_pos = (x, y)
        recompensa = -0.05  # Costo base por movimiento (fomenta eficiencia)
        terminado = False

        # Procesar interacciones con elementos del entorno
        if self.agent_pos in self.venenos:
            # Penalización por tocar veneno y reset a posición inicial
            recompensa = -10.0
            self.agent_pos = (0, 0)  # Reset automático a origen

        elif self.agent_pos in self.frutas:
            # Recompensa por recolectar fruta
            recompensa = 1.0
            self.frutas.remove(self.agent_pos)  # Eliminar fruta recolectada

            # Verificar si se completó el nivel
            if not self.frutas:  # No quedan frutas
                recompensa += 10.0   # Bonus por completar
                terminado = True     # Episodio exitoso
                self.agent_pos = (0, 0)  # Reset para próximo episodio

        return self.get_state(), recompensa, terminado

    def get_state(self):
        """
        Convierte el estado actual del entorno a formato tensor para DQN.

        Esta función es crucial para la compatibilidad con redes neuronales
        convolucionales, transformando la representación discreta del entorno
```

```
        en un tensor 3D que puede ser procesado eficientemente por la CNN.

        Returns:
            np.array: Tensor 3D con forma (3, size, size) donde:
                - Canal 0: Posición del agente (1.0 donde está, 0.0 resto)
                - Canal 1: Posiciones de frutas (1.0 donde hay frutas)
                - Canal 2: Posiciones de venenos (1.0 donde hay venenos)

        Características del formato:
        - Tipo float32 para compatibilidad con PyTorch
        - Representación binaria (0.0 o 1.0) para claridad
        - Canales separados permiten que la CNN detecte patrones específicos
        - Dimensiones compatibles con arquitectura Conv2D

        Nota: Las paredes no se incluyen en el estado ya que son estáticas
              y el agente las aprende a través de las restricciones de movimiento.
        """
        # Inicializar tensor de estado con ceros
        estado = np.zeros((3, self.size, self.size), dtype=np.float32)

        # Canal 0: Posición del agente
        estado[0, self.agent_pos[0], self.agent_pos[1]] = 1.0

        # Canal 1: Posiciones de frutas
        for fruta in self.frutas:
            estado[1, fruta[0], fruta[1]] = 1.0

        # Canal 2: Posiciones de venenos
        for veneno in self.venenos:
            estado[2, veneno[0], veneno[1]] = 1.0

        return estado

    def dibujar(
        self,
        pantalla,
        modo_juego,
        cursor_pos,
        img_fruta,
        img_veneno,
        img_pared,
        img_agente,
```

```
    ):
        """
        Renderiza el entorno completo en la pantalla usando Pygame.

        Esta función maneja toda la visualización del entorno, incluyendo
        elementos del juego, interfaz de usuario y información contextual.

        Args:
            pantalla (pygame.Surface): Superficie donde renderizar
            modo_juego (str): Modo actual ("SETUP" o "PLAYING")
            cursor_pos (tuple): Posición del cursor en modo setup
            img_fruta (pygame.Surface): Imagen para representar frutas
            img_veneno (pygame.Surface): Imagen para representar venenos
            img_pared (pygame.Surface): Imagen para representar paredes
            img_agente (pygame.Surface): Imagen para representar al agente

        Proceso de renderizado:
        1. Limpiar pantalla con color de fondo
        2. Dibujar grilla de referencia
        3. Renderizar elementos por capas (paredes → frutas → venenos → agente)
        4. Mostrar cursor en modo setup
        5. Renderizar información de controles y estado

        El orden de renderizado es importante para la superposición correcta
        de elementos visuales y la legibilidad de la interfaz.
        """
        # Limpiar pantalla con color de fondo
        pantalla.fill(COLOR_FONDO)

        # Dibujar grilla de referencia
        # Líneas verticales
        for x in range(0, SCREEN_WIDTH, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (x, 0), (x, SCREEN_HEIGHT))
        # Líneas horizontales
        for y in range(0, SCREEN_HEIGHT, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (0, y), (SCREEN_WIDTH, y))

        # Renderizar elementos del entorno (orden de capas importante)
        # 1. Paredes (fondo) - obstáculos estáticos
        for pared in self.paredes:
            pantalla.blit(img_pared, (pared[0] * CELL_SIZE, pared[1] * CELL_SIZE))
```

```
        # 2. Frutas (objetivos) - elementos a recolectar
        for fruta in self.frutas:
            pantalla.blit(img_fruta, (fruta[0] * CELL_SIZE, fruta[1] * CELL_SIZE))

        # 3. Venenos (peligros) - elementos a evitar
        for veneno in self.venenos:
            pantalla.blit(img_veneno, (veneno[0] * CELL_SIZE, veneno[1] * CELL_SIZE))

        # 4. Agente (primer plano) - jugador controlado por IA
        pantalla.blit(
            img_agente, (self.agent_pos[0] * CELL_SIZE, self.agent_pos[1] * CELL_SIZE)
        )

        # 5. Cursor de selección (solo en modo setup)
        if modo_juego == "SETUP":
            cursor_rect = pygame.Rect(
                cursor_pos[0] * CELL_SIZE,
                cursor_pos[1] * CELL_SIZE,
                CELL_SIZE,
                CELL_SIZE,
            )
            pygame.draw.rect(pantalla, COLOR_CURSOR, cursor_rect, 3)

        # Renderizar información textual de la interfaz
        font = pygame.font.Font(None, 24)

        # Mostrar modo actual
        texto_modo = font.render(f"Modo: {modo_juego}", True, COLOR_TEXTO)

        # Instrucciones para modo setup
        controles1 = font.render(
            "SETUP: Flechas, F=Fruta, V=Veneno, W=Pared, C=Limpiar", True, COLOR_TEXTO
        )

        # Controles generales
        controles2 = font.render("P=Jugar, S=Setup", True, COLOR_TEXTO)

        # Posicionar textos en la parte inferior de la pantalla
        pantalla.blit(texto_modo, (10, SCREEN_HEIGHT + 5))
        pantalla.blit(controles1, (10, SCREEN_HEIGHT + 30))
        pantalla.blit(controles2, (10, SCREEN_HEIGHT + 55))
```

```
def main():
    """
    Función principal que ejecuta la interfaz de demostración DQN.

    Esta función implementa un sistema completo de demostración interactiva
    que permite a los usuarios configurar entornos personalizados y observar
    el comportamiento de un agente DQN entrenado.

    Flujo de la aplicación:
    1. Inicialización de Pygame y carga de recursos visuales
    2. Carga del agente DQN preentrenado
    3. Bucle principal con dos modos de operación:
       - SETUP: Configuración interactiva del entorno
       - PLAYING: Demostración del agente entrenado
    4. Renderizado continuo y gestión de eventos

    Modos de operación:

    **MODO SETUP (Configuración):**
    - Navegación con flechas del teclado
    - F: Añadir/quitar frutas en posición del cursor
    - V: Añadir/quitar venenos en posición del cursor
    - W: Añadir/quitar paredes en posición del cursor
    - C: Limpiar completamente el entorno

    **MODO PLAYING (Demostración):**
    - El agente DQN toma control automático
    - Visualización en tiempo real de decisiones
    - Finalización automática y retorno a setup

    **Controles Globales:**
    - P: Cambiar a modo PLAYING
    - S: Cambiar a modo SETUP
    - ESC/X: Salir de la aplicación
    """
    # Inicializar sistema gráfico Pygame
    pygame.init()
    pantalla = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
    pygame.display.set_caption("Agente DQN - Come Frutas ")

    def cargar_img(nombre, color_fallback):
        """
```

```
        Función auxiliar para cargar imágenes con respaldo de color.

        Intenta cargar una imagen desde archivo y, si falla, crea una
        superficie de color sólido como alternativa. Esto garantiza que
        la aplicación funcione incluso sin los archivos de imagen.

        Args:
            nombre (str): Nombre/ruta del archivo de imagen
            color_fallback (tuple): Color RGB de respaldo (r, g, b)

        Returns:
            pygame.Surface: Superficie escalada al tamaño de celda
        """
        try:
            ruta = os.path.join(os.path.dirname(__file__), nombre)
            img = pygame.image.load(ruta).convert_alpha()
            return pygame.transform.scale(img, (CELL_SIZE, CELL_SIZE))
        except:
            # Crear superficie de color sólido como respaldo
            surf = pygame.Surface((CELL_SIZE, CELL_SIZE))
            surf.fill(color_fallback)
            return surf

    # Cargar recursos visuales con colores de respaldo
    img_fruta = cargar_img("../fruta.png", (0, 255, 0))      # Verde si falla
    img_veneno = cargar_img("../veneno.png", (255, 0, 0))     # Rojo si falla
    img_pared = cargar_img("../pared.png", (100, 100, 100))   # Gris si falla
    img_agente = cargar_img("../agente.png", (0, 0, 255))     # Azul si falla

    # Inicializar componentes principales
    entorno = EntornoGrid()                                    # Entorno de simulación
    agente = Agent(state_shape=(3, GRID_HEIGHT, GRID_WIDTH), action_size=4)  # Agente DQN
    agente.load("DQN/dqn_model.pth")                           # Cargar modelo preentrenado

    # Variables de control de la interfaz
    cursor_pos = [0, 0]        # Posición del cursor en modo setup
    modo_juego = "SETUP"       # Modo inicial
    reloj = pygame.time.Clock()  # Control de framerate
    corriendo = True           # Flag principal del bucle

    # Bucle principal de la aplicación
    while corriendo:
```

```
        # Procesar eventos del usuario
        for evento in pygame.event.get():
            if evento.type == pygame.QUIT:
                corriendo = False

            if evento.type == pygame.KEYDOWN:
                # --- CONTROLES GLOBALES ---
                if evento.key == pygame.K_p:
                    print("--- MODO JUEGO ---")
                    entorno.reset_a_configuracion_inicial()
                    modo_juego = "PLAYING"
                    time.sleep(0.5)  # Pausa para evitar acciones inmediatas

                elif evento.key == pygame.K_s:
                    print("--- MODO SETUP ---")
                    modo_juego = "SETUP"

                # --- CONTROLES ESPECÍFICOS DEL MODO SETUP ---
                if modo_juego == "SETUP":
                    # Navegación del cursor con flechas del teclado
                    if evento.key == pygame.K_UP:
                        cursor_pos[1] = max(0, cursor_pos[1] - 1)
                    elif evento.key == pygame.K_DOWN:
                        cursor_pos[1] = min(GRID_HEIGHT - 1, cursor_pos[1] + 1)
                    elif evento.key == pygame.K_LEFT:
                        cursor_pos[0] = max(0, cursor_pos[0] - 1)
                    elif evento.key == pygame.K_RIGHT:
                        cursor_pos[0] = min(GRID_WIDTH - 1, cursor_pos[0] + 1)

                    # Obtener posición actual del cursor
                    pos = tuple(cursor_pos)

                    # Gestión de elementos en la posición del cursor
                    if evento.key == pygame.K_f:
                        # Alternar fruta en posición actual
                        if pos in entorno.frutas:
                            entorno.frutas.remove(pos)
                        else:
                            entorno.frutas.add(pos)
                            # Limpiar otros elementos de la misma posición
                            entorno.venenos.discard(pos)
                            entorno.paredes.discard(pos)
```

```
                elif evento.key == pygame.K_v:
                    # Alternar veneno en posición actual
                    if pos in entorno.venenos:
                        entorno.venenos.remove(pos)
                    else:
                        entorno.venenos.add(pos)
                        # Limpiar otros elementos de la misma posición
                        entorno.frutas.discard(pos)
                        entorno.paredes.discard(pos)

                elif evento.key == pygame.K_w:
                    # Alternar pared en posición actual
                    if pos in entorno.paredes:
                        entorno.paredes.remove(pos)
                    else:
                        entorno.paredes.add(pos)
                        # Limpiar otros elementos de la misma posición
                        entorno.frutas.discard(pos)
                        entorno.venenos.discard(pos)

                elif evento.key == pygame.K_c:
                    # Limpiar completamente el entorno
                    print("--- LIMPIANDO ENTORNO ---")
                    entorno.limpiar_entorno()

    # --- LÓGICA DEL MODO PLAYING ---
    if modo_juego == "PLAYING":
        # Obtener estado actual del entorno
        estado = entorno.get_state()

        # El agente DQN elige la mejor acción (sin exploración)
        # explore=False garantiza que use solo la política aprendida
        accion = agente.choose_action(estado, explore=False)

        # Ejecutar la acción en el entorno
        _, _, terminado = entorno.step(accion)

        # Verificar si el episodio terminó
        if terminado:
            print("Juego terminado. Volviendo a SETUP.")
            modo_juego = "SETUP"
```

```
            # Control de velocidad para observación humana
            time.sleep(0.1)  # 10 FPS para visualización clara

        # --- SISTEMA DE RENDERIZADO ---
        # Crear superficie temporal para composición
        pantalla_con_info = pygame.Surface((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
        pantalla_con_info.fill(COLOR_FONDO)

        # Renderizar el entorno completo en la superficie temporal
        entorno.dibujar(
            pantalla_con_info,
            modo_juego,
            tuple(cursor_pos),  # Convertir lista a tupla
            img_fruta,
            img_veneno,
            img_pared,
            img_agente,
        )

        # Transferir superficie temporal a pantalla principal
        pantalla.blit(pantalla_con_info, (0, 0))

        # Actualizar pantalla y controlar framerate
        pygame.display.flip()
        reloj.tick(60)  # Limitar a 60 FPS para suavidad

    # Limpiar recursos al salir de la aplicación
    pygame.quit()


if __name__ == "__main__":
    """
    Punto de entrada del programa de demostración DQN.

    Ejecuta la función main() cuando el archivo se ejecuta directamente.
    Este patrón permite importar clases y funciones de este módulo sin
    ejecutar automáticamente la interfaz de demostración.

    Uso típico:
        python dqn_agente_comefrutas.py  # Ejecuta la demostración

    La aplicación está diseñada para:
```

```
    - Demostraciones educativas de algoritmos DQN
    - Validación visual del comportamiento del agente
    - Experimentación rápida con configuraciones de entorno
    - Presentaciones de proyectos de IA/ML

    Diferencias con versión DDQN:
    - Utiliza algoritmo DQN clásico (una sola red)
    - Compatible con modelos entrenados con DQN simple
    - Interfaz idéntica pero agente subyacente diferente
    """
    main()
```

enviroment.py

```
# environment.py
"""
Entorno de cuadrícula para el entrenamiento de un agente DQN.

Este módulo implementa un entorno de juego donde un agente debe navegar
por una cuadrícula para recoger frutas mientras evita venenos. El entorno
utiliza reward shaping para guiar al agente hacia las frutas.

"""

import numpy as np

class GridEnvironment:
    """
    Entorno de cuadrícula para un agente que debe recoger frutas y evitar venenos.

    El entorno consiste en una cuadrícula de tamaño configurable donde:
    - El agente se mueve en 4 direcciones (arriba, abajo, izquierda, derecha)
    - Las frutas proporcionan recompensas positivas
    - Los venenos proporcionan recompensas negativas y terminan el juego
    - El objetivo es recoger todas las frutas sin tocar venenos

    Attributes:
        size (int): Tamaño de la cuadrícula (size x size)
        agent_pos (np.array): Posición actual del agente [fila, columna]
        fruit_pos (list): Lista de posiciones de frutas
        poison_pos (list): Lista de posiciones de venenos
```

```
    """
    def __init__(self, size=5):
        """
        Inicializa el entorno de cuadrícula.

        Args:
            size (int, optional): Tamaño de la cuadrícula. Por defecto es 5x5.
        """
        self.size = size
        self.reset()

    def reset(self, agent_pos=(0, 0), fruit_pos=[], poison_pos=[]):
        """
        Reinicia el entorno con una configuración específica.

        Establece las posiciones iniciales del agente, frutas y venenos.
        Si no se proporcionan posiciones, se usan listas vacías para frutas y venenos.

        Args:
            agent_pos (tuple, optional): Posición inicial del agente (fila, columna).
                                         Por defecto (0, 0).
            fruit_pos (list, optional): Lista de tuplas con posiciones de frutas.
                                        Por defecto lista vacía.
            poison_pos (list, optional): Lista de tuplas con posiciones de venenos.
                                         Por defecto lista vacía.

        Returns:
            np.array: Estado inicial del entorno como array 3D (3, size, size).
        """
        self.agent_pos = np.array(agent_pos)
        self.fruit_pos = [np.array(p) for p in fruit_pos]
        self.poison_pos = [np.array(p) for p in poison_pos]
        return self.get_state()

    def get_state(self):
        """
        Genera la representación del estado actual del entorno.

        El estado se representa como una "imagen" de 3 canales que puede ser
        procesada por una CNN. Cada canal representa un tipo de elemento:

        - Canal 0: Posición del agente (1.0 donde está el agente, 0.0 en el resto)
```

```
        - Canal 1: Posiciones de frutas (1.0 donde hay frutas, 0.0 en el resto)
        - Canal 2: Posiciones de venenos (1.0 donde hay venenos, 0.0 en el resto)

        Esta representación permite que el agente "vea" todo el entorno de una vez
        y facilita el procesamiento por redes neuronales convolucionales.

        Returns:
            np.array: Estado del entorno como array 3D de forma (3, size, size)
                      con valores float32.
        """
        state = np.zeros((3, self.size, self.size), dtype=np.float32)

        # Canal 0: Posición del agente
        state[0, self.agent_pos[0], self.agent_pos[1]] = 1.0

        # Canal 1: Posiciones de las frutas
        for fruit in self.fruit_pos:
            state[1, fruit[0], fruit[1]] = 1.0

        # Canal 2: Posiciones de los venenos
        for poison in self.poison_pos:
            state[2, poison[0], poison[1]] = 1.0

        return state

    def step(self, action):
        """
        Ejecuta una acción en el entorno y retorna el resultado.

        Esta función implementa la lógica principal del juego, incluyendo:
        1. Movimiento del agente
        2. Cálculo de recompensas con reward shaping
        3. Detección de colisiones con frutas y venenos
        4. Determinación de condiciones de terminación

        El sistema de recompensas incluye:
        - Recompensa por acercarse a frutas (+0.1)
        - Castigo por alejarse de frutas (-0.15)
        - Recompensa por recoger frutas (+1.0)
        - Castigo por tocar veneno (-1.0, termina el juego)
        - Recompensa por completar el nivel (+5.0)
        - Castigo base por movimiento (-0.05, fomenta eficiencia)
```

```
        Args:
            action (int): Acción a realizar:
                          0 = Arriba (decrementar fila)
                          1 = Abajo (incrementar fila)
                          2 = Izquierda (decrementar columna)
                          3 = Derecha (incrementar columna)

        Returns:
            tuple: (nuevo_estado, recompensa, terminado)
                - nuevo_estado (np.array): Estado del entorno después de la acción
                - recompensa (float): Recompensa obtenida por la acción
                - terminado (bool): True si el episodio ha terminado
        """

        # FASE 1: REWARD SHAPING - Calcular distancia a fruta más cercana ANTES del movimient
        # Esto permite dar recompensas por acercarse/alejarse de las frutas
        old_dist_to_fruit = float('inf')
        if self.fruit_pos:
            distances = [np.linalg.norm(self.agent_pos - fruit) for fruit in self.fruit_pos]
            old_dist_to_fruit = min(distances)


        # FASE 2: MOVIMIENTO DEL AGENTE
        # Actualizar la posición del agente basada en la acción seleccionada
        if action == 0:      # Arriba
            self.agent_pos[0] -= 1
        elif action == 1:    # Abajo
            self.agent_pos[0] += 1
        elif action == 2:    # Izquierda
            self.agent_pos[1] -= 1
        elif action == 3:    # Derecha
            self.agent_pos[1] += 1

        # Limitar la posición del agente a los límites del tablero
        # np.clip asegura que las coordenadas estén entre 0 y (size-1)
        self.agent_pos = np.clip(self.agent_pos, 0, self.size - 1)


        # FASE 3: CÁLCULO DE RECOMPENSAS

        # Recompensa base: pequeño castigo por cada movimiento para fomentar eficiencia
        reward = -0.05
```

```
        done = False

        # REWARD SHAPING: Calcular nueva distancia y recompensar acercamiento a frutas
        # Esto ayuda al agente a aprender a navegar hacia las frutas incluso antes de alcanza
        new_dist_to_fruit = float('inf')
        if self.fruit_pos:
            distances = [np.linalg.norm(self.agent_pos - fruit) for fruit in self.fruit_pos]
            new_dist_to_fruit = min(distances)

            # Recompensar por acercarse, castigar por alejarse
            if new_dist_to_fruit < old_dist_to_fruit:
                reward += 0.1   # Recompensa por acercarse a una fruta
            else:
                reward -= 0.15  # Castigo por alejarse (ligeramente mayor para evitar indeci


        # FASE 4: DETECCIÓN DE EVENTOS

        # Verificar si el agente recogió una fruta
        for i, fruit in enumerate(self.fruit_pos):
            if np.array_equal(self.agent_pos, fruit):
                reward += 1.0  # Gran recompensa por recoger fruta
                self.fruit_pos.pop(i)  # Remover la fruta del entorno
                break  # Solo puede recoger una fruta por paso

        # Verificar si el agente tocó veneno (termina el juego)
        if any(np.array_equal(self.agent_pos, poison) for poison in self.poison_pos):
            reward = -1.0  # Castigo severo y absoluto por tocar veneno
            done = True    # Terminar el episodio inmediatamente

        # Verificar condición de victoria: no quedan frutas
        if not self.fruit_pos:
            done = True
            reward += 5.0  # Gran recompensa bonus por completar el objetivo

        return self.get_state(), reward, done
```

main.py

```
# main.py
"""
```

```
Interfaz gráfica interactiva para visualizar un agente DQN entrenado.

Este módulo implementa una aplicación Pygame que permite:
1. Configurar un escenario colocando frutas y venenos manualmente
2. Observar cómo el agente DQN entrenado resuelve el escenario
3. Reiniciar para probar diferentes configuraciones

La aplicación tiene dos modos:
- Modo Setup: El usuario coloca elementos en la cuadrícula
- Modo Run: El agente toma control y ejecuta su política aprendida

Controles:
- Click izquierdo: Colocar fruta
- Click derecho: Colocar veneno
- Espacio: Iniciar simulación del agente

"""

import pygame
import numpy as np
from environment import GridEnvironment
from agent import Agent

# CONFIGURACIÓN DE PYGAME Y CONSTANTES DEL JUEGO
"""
Configuración visual y dimensiones de la aplicación.
"""
GRID_SIZE = 5        # Tamaño de la cuadrícula (5x5)
CELL_SIZE = 100      # Tamaño de cada celda en píxeles
WIDTH, HEIGHT = GRID_SIZE * CELL_SIZE, GRID_SIZE * CELL_SIZE  # Ventana de 500x500 píxeles
WIN = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Agente Come-Frutas")
pygame.font.init()   # Inicializar fuentes para texto si se necesita

# INICIALIZACIÓN DEL AGENTE DQN ENTRENADO
"""
Carga el agente DQN previamente entrenado desde archivo.
El agente utilizará su política aprendida para navegar por el entorno.
"""
env = GridEnvironment(size=GRID_SIZE)
action_size = 4  # 4 acciones posibles: arriba, abajo, izquierda, derecha
state_shape = (3, GRID_SIZE, GRID_SIZE)  # Forma del estado: 3 canales x 5x5 grid
```

```
agent = Agent(state_shape, action_size)  # Crear instancia del agente
agent.load("dqn_model.pth")              # Cargar pesos del modelo entrenado

# DEFINICIÓN DE COLORES
"""
Paleta de colores para los elementos visuales del juego.
Utiliza sistema RGB (Red, Green, Blue) con valores 0-255.
"""
COLOR_GRID = (200, 200, 200)   # Gris claro para las líneas de la cuadrícula
COLOR_AGENT = (0, 0, 255)      # Azul para el agente
COLOR_FRUIT = (0, 255, 0)      # Verde para las frutas
COLOR_POISON = (255, 0, 0)     # Rojo para los venenos

def draw_grid():
    """
    Dibuja las líneas de la cuadrícula en la ventana.

    Crea una cuadrícula visual de 5x5 dibujando líneas verticales y horizontales
    separadas por CELL_SIZE píxeles. Esto ayuda a visualizar las celdas donde
    se pueden colocar elementos y donde se mueve el agente.
    """
    # Líneas verticales
    for x in range(0, WIDTH, CELL_SIZE):
        pygame.draw.line(WIN, COLOR_GRID, (x, 0), (x, HEIGHT))
    # Líneas horizontales
    for y in range(0, HEIGHT, CELL_SIZE):
        pygame.draw.line(WIN, COLOR_GRID, (0, y), (WIDTH, y))

def draw_elements(agent_pos, fruits, poisons):
    """
    Dibuja todos los elementos del juego en sus posiciones actuales.

    Renderiza visualmente:
    - Agente: Como un cuadrado azul que ocupa toda la celda
    - Frutas: Como círculos verdes centrados en sus celdas
    - Venenos: Como cuadrados rojos más pequeños centrados en sus celdas

    Args:
        agent_pos (np.array): Posición del agente [fila, columna]
        fruits (list): Lista de posiciones de frutas [(fila, col), ...]
        poisons (list): Lista de posiciones de venenos [(fila, col), ...]
```

```
    Note:
        Las coordenadas se invierten para Pygame: agent_pos[1] es X, agent_pos[0] es Y
    """
    # Dibujar agente como cuadrado azul completo
    if agent_pos[0] >= 0:  # Solo dibujar si el agente está en el tablero
        pygame.draw.rect(WIN, COLOR_AGENT,
                         (agent_pos[1] * CELL_SIZE, agent_pos[0] * CELL_SIZE,
                          CELL_SIZE, CELL_SIZE))

    # Dibujar frutas como círculos verdes
    for f in fruits:
        center_x = f[1] * CELL_SIZE + CELL_SIZE // 2
        center_y = f[0] * CELL_SIZE + CELL_SIZE // 2
        radius = CELL_SIZE // 3
        pygame.draw.circle(WIN, COLOR_FRUIT, (center_x, center_y), radius)

    # Dibujar venenos como cuadrados rojos más pequeños
    for p in poisons:
        margin = 20  # Margen para hacer el cuadrado más pequeño
        pygame.draw.rect(WIN, COLOR_POISON,
                         (p[1] * CELL_SIZE + margin, p[0] * CELL_SIZE + margin,
                          CELL_SIZE - 2*margin, CELL_SIZE - 2*margin))

def main():
    """
    Función principal que maneja el bucle de la aplicación.

    Implementa una máquina de estados con dos modos:

    MODO SETUP:
    - Permite al usuario colocar frutas y venenos con clics del mouse
    - Click izquierdo: Colocar fruta
    - Click derecho: Colocar veneno
    - Presionar ESPACIO: Iniciar simulación

    MODO RUN:
    - El agente DQN toma control del juego
    - Ejecuta acciones basadas en su política aprendida
    - Visualiza el comportamiento del agente en tiempo real
    - Se reinicia automáticamente al terminar

    La aplicación se ejecuta hasta que el usuario cierre la ventana.
```

```
    """
    # Variables de estado del juego
    fruits = []      # Lista de posiciones de frutas colocadas por el usuario
    poisons = []     # Lista de posiciones de venenos colocadas por el usuario
    mode = "setup"   # Modo actual: "setup" (configuración) o "run" (simulación)

    # Configuración del bucle principal
    clock = pygame.time.Clock()  # Para controlar FPS
    run = True                   # Flag de control del bucle principal
    # BUCLE PRINCIPAL DE LA APLICACIÓN
    while run:
        # Limpiar pantalla con fondo negro
        WIN.fill((0, 0, 0))
        # Dibujar cuadrícula base
        draw_grid()

        # MANEJO DE EVENTOS DE USUARIO
        for event in pygame.event.get():
            # Evento de cierre de ventana
            if event.type == pygame.QUIT:
                run = False

            # EVENTOS EN MODO SETUP (Configuración manual)
            if mode == "setup":
                # Manejo de clics del mouse para colocar elementos
                if event.type == pygame.MOUSEBUTTONDOWN:
                    pos = pygame.mouse.get_pos()
                    # Convertir coordenadas de píxeles a coordenadas de cuadrícula
                    col = pos[0] // CELL_SIZE
                    row = pos[1] // CELL_SIZE

                    # Click izquierdo (botón 1): Colocar fruta
                    if event.button == 1 and (row, col) not in fruits:
                        fruits.append((row, col))
                        print(f"Fruta colocada en ({row}, {col})")

                    # Click derecho (botón 3): Colocar veneno
                    elif event.button == 3 and (row, col) not in poisons:
                        poisons.append((row, col))
                        print(f"Veneno colocado en ({row}, {col})")

                # Manejo de teclas para cambiar de modo
```

```
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE:
                    mode = "run"
                    # Inicializar el entorno con la configuración del usuario
                    state = env.reset(agent_pos=(0, 0), fruit_pos=fruits, poison_pos=pois
                    print("=== INICIANDO SIMULACIÓN DEL AGENTE ===")
                    print(f"Frutas: {len(fruits)}, Venenos: {len(poisons)}")

    # RENDERIZADO SEGÚN EL MODO ACTUAL

    if mode == "setup":
        # MODO CONFIGURACIÓN: Mostrar elementos colocados por el usuario
        # Usar posición (-1,-1) para que el agente no aparezca en pantalla
        draw_elements(np.array([-1, -1]), fruits, poisons)

    elif mode == "run":
        # MODO SIMULACIÓN: El agente DQN ejecuta su política

        # Obtener estado actual del entorno
        state = env.get_state()

        # El agente decide la acción usando su política entrenada
        # explore=False significa que usa solo explotación, no exploración
        action = agent.choose_action(state, explore=False)

        # Ejecutar la acción en el entorno
        next_state, reward, done = env.step(action)

        # Renderizar estado actual del juego
        draw_elements(env.agent_pos, env.fruit_pos, env.poison_pos)

        # Verificar si el episodio terminó
        if done:
            if not env.fruit_pos:  # Victoria: todas las frutas recogidas
                print("  ¡ÉXITO! El agente recogió todas las frutas")
            else:  # Derrota: tocó veneno
                print("  DERROTA: El agente tocó veneno")

            print("=== SIMULACIÓN TERMINADA ===")

            # Reiniciar para permitir nueva configuración
            fruits = []
```

```
                poisons = []
                mode = "setup"

                # Pausa dramática antes de reiniciar
                pygame.time.delay(2000)

            # Pausa entre movimientos para visualización clara
            pygame.time.delay(300)

        # Actualizar pantalla con todos los cambios
        pygame.display.update()

    # Limpieza al cerrar la aplicación
    pygame.quit()


if __name__ == "__main__":
    """
    Punto de entrada del programa.

    Ejecuta la función main() solo si este archivo se ejecuta directamente
    (no si se importa como módulo).
    """
    print("=== AGENTE DQN COME-FRUTAS ===")
    print("CONTROLES:")
    print("• Click izquierdo: Colocar fruta")
    print("• Click derecho: Colocar veneno")
    print("• ESPACIO: Iniciar simulación")
    print("• Cerrar ventana: Salir")
    print("\n¡Configura un escenario y observa al agente!")

    main()
```

DDQN

agent.py

```
# agent.py
"""
Implementación completa del agente DDQN (Double Deep Q-Network).
```

```
Este módulo contiene la implementación del algoritmo DDQN, una mejora del DQN clásico
que aborda el problema de sobreestimación de valores Q mediante el uso de dos redes
neuronales: una para selección de acciones y otra para evaluación de valores.

Características principales:
- Red neuronal convolucional optimizada para entornos de grilla
- Algoritmo DDQN con separación de selección y evaluación
- Memoria de replay extendida (50,000 experiencias)
- Técnicas de estabilización avanzadas
- Sistema robusto de guardado/carga de modelos

Algoritmo DDQN:
La innovación clave es el uso de dos redes para calcular targets:
1. Red principal: Selecciona la mejor acción del siguiente estado
2. Red objetivo: Evalúa el valor Q de esa acción seleccionada

Esto reduce significativamente la sobreestimación de valores Q que sufre DQN clásico,
resultando en un aprendizaje más estable y políticas más robustas.

Referencias:
- van Hasselt et al. (2016): "Deep Reinforcement Learning with Double Q-learning"
- Mnih et al. (2015): "Human-level control through deep reinforcement learning"
"""

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
from collections import deque

# --- 1. RED NEURONAL CONVOLUCIONAL PARA DDQN ---
class CNN_DQN(nn.Module):
    """
    Red neuronal convolucional especializada para DDQN en entornos de grilla.

    Esta arquitectura está optimizada para procesar estados representados como
    imágenes multi-canal, típicos en problemas de navegación espacial donde
    el estado se puede visualizar como una cuadrícula con diferentes tipos
    de elementos (agente, objetivos, obstáculos).

    Diseño arquitectónico:
```

```
    **Etapa Convolucional (Extracción de características):**
    - Conv1: 3→16 canales, kernel 3x3 → Detecta patrones básicos locales
    - Conv2: 16→32 canales, kernel 3x3 → Combina patrones en características complejas
    - ReLU en cada capa para introducir no-linealidad
    - Padding=1 preserva dimensiones espaciales

    **Etapa Completamente Conectada (Toma de decisiones):**
    - FC1: Procesa características extraídas (256 neuronas)
    - FC2: Genera valores Q para cada acción posible

    **Ventajas de esta arquitectura:**
    - Invarianza a traslaciones locales (convoluciones)
    - Reducción progresiva de parámetros vs redes totalmente conectadas
    - Capacidad de detectar patrones espaciales complejos
    - Escalabilidad a entornos de diferentes tamaños

    Args:
        h (int): Altura de la grilla de entrada
        w (int): Anchura de la grilla de entrada
        outputs (int): Número de acciones posibles (valores Q de salida)
    """

    def __init__(self, h, w, outputs):
        super(CNN_DQN, self).__init__()

        # --- CAPAS CONVOLUCIONALES PARA EXTRACCIÓN DE CARACTERÍSTICAS ---
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        # Entrada: 3 canales (agente, frutas, venenos)
        # Salida: 16 mapas de características
        # Kernel 3x3: Ventana de percepción local óptima para grillas pequeñas
        # Padding=1: Preserva dimensiones espaciales de entrada

        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        # Entrada: 16 mapas de características de la capa anterior
        # Salida: 32 mapas de características más abstractas
        # Mayor profundidad permite detectar patrones más complejos

        # --- CÁLCULO DINÁMICO DE DIMENSIONES ---
        """
        Función auxiliar para calcular dimensiones después de convoluciones.
        Esencial para conectar correctamente las capas convolucionales
        con las capas completamente conectadas.
```

```
        Fórmula: output_size = (input_size + 2*padding - kernel_size) // stride + 1
        """
        def conv2d_size_out(size, kernel_size=3, stride=1, padding=1):
            return (size + 2 * padding - kernel_size) // stride + 1

        # Aplicar la función de cálculo a ambas dimensiones espaciales
        convw = conv2d_size_out(conv2d_size_out(w))
        convh = conv2d_size_out(conv2d_size_out(h))
        linear_input_size = convw * convh * 32  # 32 canales de la última conv

        # --- CAPAS COMPLETAMENTE CONECTADAS PARA TOMA DE DECISIONES ---
        self.fc1 = nn.Linear(linear_input_size, 256)
        # Capa oculta densa que procesa las características extraídas
        # 256 neuronas: Balance entre capacidad expresiva y eficiencia computacional
        # Suficiente para capturar relaciones complejas entre características espaciales

        self.fc2 = nn.Linear(256, outputs)
        # Capa de salida que produce valores Q para cada acción
        # Sin función de activación (los valores Q pueden ser negativos)
        # Número de neuronas = número de acciones posibles

    def forward(self, x):
        """
        Propagación hacia adelante de la red neuronal.

        Implementa el flujo completo de información desde el estado de entrada
        hasta los valores Q de salida, aplicando las transformaciones necesarias
        para extraer características espaciales y generar estimaciones de valor.

        Args:
            x (torch.Tensor): Estado de entrada con forma (batch_size, 3, height, width)
                              - batch_size: Número de estados en el lote
                              - 3: Canales (agente, frutas, venenos)
                              - height, width: Dimensiones espaciales de la grilla

        Returns:
            torch.Tensor: Valores Q para cada acción con forma (batch_size, num_actions)

        Flujo de procesamiento:
        1. Convolución 1 + ReLU: Detección de patrones básicos
        2. Convolución 2 + ReLU: Extracción de características complejas
        3. Aplanamiento: Conversión de 2D a 1D para capas densas
```

```
        4. FC1 + ReLU: Procesamiento de alto nivel de características
        5. FC2: Generación de valores Q finales (sin activación)
        """
        # Primera capa convolucional con activación ReLU
        x = nn.functional.relu(self.conv1(x))

        # Segunda capa convolucional con activación ReLU
        x = nn.functional.relu(self.conv2(x))

        # Aplanar características espaciales para capas densas
        # Transforma tensor 4D (batch, canales, alto, ancho) → 2D (batch, características)
        x = x.view(x.size(0), -1)

        # Primera capa completamente conectada con activación ReLU
        x = nn.functional.relu(self.fc1(x))

        # Capa de salida sin activación (valores Q pueden ser negativos)
        return self.fc2(x)

# --- 2. AGENTE DDQN CON EXPERIENCE REPLAY Y TARGET NETWORK ---

class Agent:
    """
    Agente de Deep Q-Learning con arquitectura CNN para navegación en grilla.

    Implementa un agente de aprendizaje por refuerzo que utiliza una red neuronal
    convolucional para procesar estados espaciales y aprender una política óptima
    para navegar en un entorno de grilla, evitando venenos y recolectando frutas.

    Características principales:
    - CNN para procesamiento de estados espaciales (grilla 5x5)
    - Experience replay con buffer de memoria para estabilidad
    - Target network para cálculos de valores Q objetivo
    - Estrategia epsilon-greedy con decaimiento para exploración
    - Optimización Adam para entrenamiento eficiente

    Arquitectura del agente:
    1. Red principal: Entrenamiento y selección de acciones
    2. Red objetivo: Cálculos estables de valores Q futuro
    3. Buffer de experiencias: Almacena transiciones para replay
    4. Optimizador: Adam para actualización de pesos
```

```
    El agente mejora mediante:
    - Exploración inicial alta (epsilon=1.0) para descubrir el entorno
    - Decaimiento gradual hacia explotación (epsilon_min=0.01)
    - Entrenamiento con experiencias pasadas (experience replay)
    - Actualización periódica de la red objetivo para estabilidad
    """

    def __init__(self, state_shape, action_size):
        """
        Inicializa el agente DDQN con configuración optimizada para el entorno.

        Args:
            state_shape (tuple): Forma del estado (canales, altura, ancho)
                                 Típicamente (3, 5, 5) para grilla con agente/frutas/venenos
            action_size (int): Número de acciones posibles (4: arriba, abajo, izq, der)

        Configuración de hiperparámetros:
        - memory: 50,000 experiencias para diversidad y estabilidad
        - gamma: 0.99 (alta importancia a recompensas futuras)
        - epsilon: 1.0→0.01 (exploración total a mínima)
        - epsilon_decay: 0.9995 (decaimiento gradual)
        - learning_rate: 0.0001 (ajuste fino y estable)
        - update_target_every: 5 (frecuencia de actualización de red objetivo)
        """
        self.state_shape = state_shape
        self.action_size = action_size

        # --- CONFIGURACIÓN DE EXPERIENCE REPLAY ---
        self.memory = deque(maxlen=50000)
        # Buffer circular que almacena hasta 50,000 experiencias
        # Tamaño grande permite mayor diversidad de experiencias
        # Memoria circular: experiencias antiguas se eliminan automáticamente

        # --- PARÁMETROS DE APRENDIZAJE ---
        self.gamma = 0.99
        # Factor de descuento alto para valorar recompensas futuras
        # 0.99 significa que recompensas 100 pasos adelante valen ~37% del valor actual

        # --- ESTRATEGIA DE EXPLORACIÓN EPSILON-GREEDY ---
        self.epsilon = 1.0
        # Exploración inicial: 100% acciones aleatorias para descubrir entorno
```

```
        self.epsilon_min = 0.01
        # Exploración mínima: siempre mantener 1% de acciones aleatorias
        # Evita quedar atrapado en mínimos locales

        self.epsilon_decay = 0.9995
        # Decaimiento gradual: epsilon *= 0.9995 cada episodio
        # Transición suave de exploración a explotación

        # --- OPTIMIZACIÓN ---
        self.learning_rate = 0.0001
        # Tasa de aprendizaje baja para entrenamiento estable y convergencia suave
        # Evita oscilaciones en la función de pérdida

        # --- ACTUALIZACIÓN DE RED OBJETIVO ---
        self.update_target_every = 5
        # Frecuencia de actualización de la red objetivo (cada 5 entrenamientos)
        # Balance entre estabilidad y adaptación a nuevos pesos

        # --- INICIALIZACIÓN DE REDES NEURONALES ---
        h, w = state_shape[1], state_shape[2]  # Dimensiones de la grilla

        # Red principal: Se entrena continuamente con nuevas experiencias
        self.model = CNN_DQN(h, w, action_size)

        # Red objetivo: Proporciona valores Q estables para cálculos de objetivo
        self.target_model = CNN_DQN(h, w, action_size)

        # Inicializar red objetivo con mismos pesos que red principal
        self.update_target_network()

        # --- CONFIGURACIÓN DE ENTRENAMIENTO ---
        # Optimizador Adam: Adaptativo, eficiente para redes neuronales
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.learning_rate)

        # Función de pérdida: Error cuadrático medio para regresión de valores Q
        self.criterion = nn.MSELoss()

        # Contador de pasos para tracking de actualizaciones
        self.steps_done = 0

    def update_target_network(self):
        """
```

```
        Actualiza la red objetivo copiando pesos de la red principal.

        La red objetivo es fundamental para la estabilidad del entrenamiento:
        - Proporciona valores Q estables para calcular objetivos
        - Se actualiza menos frecuentemente que la red principal
        - Evita que los objetivos cambien constantemente durante entrenamiento

        Proceso:
        1. Copia completa de todos los parámetros de la red principal
        2. La red objetivo permanece fija hasta la próxima actualización
        3. Garantiza consistencia en los cálculos de valores Q objetivo
        """
        self.target_model.load_state_dict(self.model.state_dict())

    def remember(self, state, action, reward, next_state, done):
        """
        Almacena una experiencia en el buffer de memory replay.

        Experience replay es una técnica fundamental en Deep Q-Learning que:
        - Rompe correlaciones temporales entre experiencias consecutivas
        - Permite reutilizar experiencias valiosas múltiples veces
        - Mejora la eficiencia de uso de datos
        - Estabiliza el entrenamiento de la red neuronal

        Args:
            state (np.array): Estado actual del agente en la grilla
                              Forma (3, 5, 5) con canales para agente/frutas/venenos
            action (int): Acción tomada (0=arriba, 1=abajo, 2=izq, 3=der)
            reward (float): Recompensa recibida por la acción
                            +10 por fruta, -10 por veneno, -1 por movimiento
            next_state (np.array): Estado resultante después de la acción
            done (bool): True si el episodio terminó (todas frutas recogidas)

        El buffer circular (deque) gestiona automáticamente:
        - Eliminación de experiencias antiguas cuando se alcanza el límite
        - Mantener diversidad de experiencias para entrenamiento robusto
        - Acceso eficiente para muestreo aleatorio durante replay
        """
        self.memory.append((state, action, reward, next_state, done))

    def choose_action(self, state, explore=True):
        """
```

```
        Selecciona una acción usando estrategia epsilon-greedy.

        Implementa el balance crítico entre exploración y explotación:
        - Exploración: Necesaria para descubrir nuevas estrategias
        - Explotación: Usar conocimiento actual para maximizar recompensas

        Args:
            state (np.array): Estado actual del entorno (3, 5, 5)
            explore (bool): Si False, siempre usa la mejor acción conocida
                            Útil para evaluación sin exploración aleatoria

        Returns:
            int: Índice de acción seleccionada (0-3)

        Estrategia epsilon-greedy:
        - Probabilidad epsilon: Acción aleatoria (exploración)
        - Probabilidad (1-epsilon): Mejor acción según red neuronal (explotación)

        Progresión de epsilon:
        - Inicio:  =1.0 → 100% exploración para mapear el entorno
        - Medio:  ~0.5 → Balance exploración/explotación
        - Final:  =0.01 → 99% explotación, 1% exploración residual
        """
        self.steps_done += 1  # Contador para tracking de progreso

        # Exploración: acción aleatoria si epsilon lo determina y explore=True
        if explore and np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)

        # Explotación: usar red neuronal para encontrar mejor acción
        # Convertir estado a tensor PyTorch y agregar dimensión de batch
        state_tensor = torch.FloatTensor(state).unsqueeze(0)

        # Inferencia sin calcular gradientes (más eficiente)
        with torch.no_grad():
            action_values = self.model(state_tensor)

        # Seleccionar acción con mayor valor Q predicho
        return np.argmax(action_values.cpu().data.numpy())

    def replay(self, batch_size):
        """
```

```
        Entrena la red neuronal usando experiencias pasadas con Double DQN.

        Double DQN mejora el algoritmo DQN clásico al separar la selección
        y evaluación de acciones, reduciendo la sobrestimación sistemática
        de valores Q que puede llevar a políticas subóptimas.

        Diferencias DQN vs Double DQN:

        DQN Clásico:
        target = reward + gamma * max(target_network(next_state))
        Problema: La misma red selecciona y evalúa → sobrestimación

        Double DQN:
        best_action = argmax(main_network(next_state))     # Selección
        target = reward + gamma * target_network(next_state)[best_action]  # Evaluación
        Ventaja: Separación reduce sesgo de sobrestimación

        Args:
            batch_size (int): Tamaño del lote de experiencias para entrenamiento
                              Típicamente 32-64 para balance eficiencia/estabilidad

        Proceso de entrenamiento:
        1. Verificar que hay suficientes experiencias en memoria
        2. Muestrear batch aleatorio de experiencias
        3. Calcular valores Q actuales para estados del batch
        4. Aplicar lógica Double DQN para calcular objetivos
        5. Computar loss (MSE entre predicciones y objetivos)
        6. Backpropagation y actualización de pesos
        7. Decrecer epsilon (menos exploración)
        8. Aplicar clipping de gradientes para estabilidad
        """
        # No entrenar si memoria insuficiente
        if len(self.memory) < batch_size:
            return

        # --- MUESTREO ALEATORIO DE EXPERIENCIAS ---
        # Rompe correlaciones temporales y mejora generalización
        minibatch = random.sample(self.memory, batch_size)

        # Separar componentes de las experiencias en tensores
        states = torch.FloatTensor(np.array([e[0] for e in minibatch]))
        actions = torch.LongTensor([e[1] for e in minibatch]).unsqueeze(1)
```

```
        rewards = torch.FloatTensor([e[2] for e in minibatch]).unsqueeze(1)
        next_states = torch.FloatTensor(np.array([e[3] for e in minibatch]))
        dones = torch.BoolTensor([e[4] for e in minibatch]).unsqueeze(1)

        # --- VALORES Q ACTUALES ---
        # Calcular Q-values para estados actuales usando red principal
        current_q_values = self.model(states).gather(1, actions)

        # --- LÓGICA DOUBLE DQN ---
        with torch.no_grad():  # No calcular gradientes para eficiencia
            # 1. Red principal SELECCIONA mejor acción para siguiente estado
            #    Usa conocimiento más actualizado para selección
            best_next_actions = self.model(next_states).max(1)[1].unsqueeze(1)

            # 2. Red objetivo EVALÚA el valor de la acción seleccionada
            #    Usa pesos más estables para evaluación consistente
            next_q_values_target = self.target_model(next_states).gather(1, best_next_actions

        # --- CÁLCULO DE OBJETIVOS Q ---
        # Si episodio terminó (done=True), no hay valor futuro
        # target = reward + descuento * valor_futuro * (no_terminado)
        target_q_values = rewards + (self.gamma * next_q_values_target * (~dones))

        # --- ENTRENAMIENTO DE LA RED ---
        # Error cuadrático medio entre predicciones y objetivos
        loss = self.criterion(current_q_values, target_q_values)

        # Optimización con backpropagation
        self.optimizer.zero_grad()  # Limpiar gradientes previos
        loss.backward()             # Calcular gradientes

        # Clipping de gradientes para prevenir explosión
        # Limita gradientes a [-1, 1] para estabilidad numérica
        torch.nn.utils.clip_grad_value_(self.model.parameters(), 1)

        self.optimizer.step()       # Actualizar pesos

        # --- DECAIMIENTO DE EXPLORACIÓN ---
        # Reducir epsilon gradualmente para transición exploración→explotación
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

```
    def load(self, name):
        """
        Carga un modelo pre-entrenado desde archivo.

        Args:
            name (str): Ruta al archivo .pth con los pesos del modelo

        Funcionalidad:
        - Carga pesos de la red principal desde archivo
        - Actualiza red objetivo para mantener consistencia
        - Permite continuar entrenamiento o hacer inferencia
        - Preserva arquitectura de red definida en __init__
        """
        self.model.load_state_dict(torch.load(name))
        self.update_target_network()  # Sincronizar red objetivo

    def save(self, name):
        """
        Guarda el modelo entrenado en un archivo.

        Args:
            name (str): Ruta donde guardar el archivo .pth

        Funcionalidad:
        - Guarda solo los pesos de la red principal (más compacto)
        - La red objetivo se puede reconstruir al cargar
        - Formato PyTorch estándar para compatibilidad
        - Permite reutilizar modelos entrenados
        """
        torch.save(self.model.state_dict(), name)
```

enviroment.py

```
# environment.py
"""
Entorno de grilla para el entrenamiento de agentes de aprendizaje por refuerzo.
Este módulo implementa un entorno de grilla donde un agente debe recolectar frutas
mientras evita venenos, diseñado específicamente para algoritmos DDQN.
"""

import numpy as np
```

```python
class GridEnvironment:
    """
    Entorno de grilla 2D para simulación de agentes que recolectan frutas y evitan venenos.

    El entorno consiste en una grilla cuadrada donde:
    - El agente se mueve en 4 direcciones (arriba, abajo, izquierda, derecha)
    - Las frutas otorgan recompensas positivas cuando son recolectadas
    - Los venenos causan penalizaciones y resetean la posición del agente
    - El objetivo es recolectar todas las frutas minimizando las penalizaciones

    Attributes:
        size (int): Tamaño de la grilla (size x size)
        start_pos (tuple): Posición inicial del agente en cada episodio
        agent_pos (np.array): Posición actual del agente
        fruit_pos (list): Lista de posiciones de las frutas
        poison_pos (list): Lista de posiciones de los venenos
    """

    def __init__(self, size=5):
        """
        Inicializa el entorno de grilla.

        Args:
            size (int, optional): Tamaño de la grilla cuadrada. Por defecto es 5x5.
        """
        self.size = size
        self.start_pos = (0, 0)  # Posición inicial por defecto
        self.reset()

    def reset(self, agent_pos=(0, 0), fruit_pos=[], poison_pos=[]):
        """
        Reinicia el entorno con una configuración específica.

        Este método prepara el entorno para un nuevo episodio, estableciendo las posiciones
        iniciales del agente, frutas y venenos. Es crucial para el entrenamiento ya que
        permite configurar diferentes escenarios de aprendizaje.

        Args:
            agent_pos (tuple, optional): Posición inicial del agente (x, y). Por defecto (0,
            fruit_pos (list, optional): Lista de tuplas con posiciones de frutas. Por defecto
            poison_pos (list, optional): Lista de tuplas con posiciones de venenos. Por defec
```

```
        Returns:
            np.array: Estado inicial del entorno como tensor 3D (canales, altura, anchura).
        """
        self.start_pos = np.array(agent_pos)  # Guardamos la posición inicial del episodio
        self.agent_pos = np.array(agent_pos)
        self.fruit_pos = [np.array(p) for p in fruit_pos]
        self.poison_pos = [np.array(p) for p in poison_pos]
        return self.get_state()

    def get_state(self):
        """
        Obtiene el estado actual del entorno como una representación tensorial.

        El estado se representa como un tensor 3D de forma (3, size, size) donde:
        - Canal 0: Posición del agente (1.0 en la posición actual, 0.0 en el resto)
        - Canal 1: Posiciones de las frutas (1.0 donde hay frutas, 0.0 en el resto)
        - Canal 2: Posiciones de los venenos (1.0 donde hay venenos, 0.0 en el resto)

        Esta representación permite que las redes neuronales procesen eficientemente
        la información espacial del entorno usando convoluciones.

        Returns:
            np.array: Tensor 3D de forma (3, size, size) representando el estado actual.
        """
        state = np.zeros((3, self.size, self.size), dtype=np.float32)

        # Canal 0: Posición del agente
        state[0, self.agent_pos[0], self.agent_pos[1]] = 1.0

        # Canal 1: Posiciones de las frutas
        for fruit in self.fruit_pos:
            state[1, fruit[0], fruit[1]] = 1.0

        # Canal 2: Posiciones de los venenos
        for poison in self.poison_pos:
            state[2, poison[0], poison[1]] = 1.0

        return state

    def step(self, action):
        """
        Ejecuta una acción en el entorno y devuelve el resultado.
```

```
        Este método implementa la lógica principal del entorno, procesando las acciones
        del agente y calculando las recompensas correspondientes. Incluye manejo especial
        para venenos que resetean la posición del agente sin terminar el episodio.

        Args:
            action (int): Acción a ejecutar
                - 0: Mover hacia arriba (decrementar fila)
                - 1: Mover hacia abajo (incrementar fila)
                - 2: Mover hacia la izquierda (decrementar columna)
                - 3: Mover hacia la derecha (incrementar columna)

        Returns:
            tuple: (nuevo_estado, recompensa, episodio_terminado)
                - nuevo_estado (np.array): Estado resultante después de la acción
                - recompensa (float): Recompensa obtenida por la acción
                - episodio_terminado (bool): True si el episodio ha terminado

        Lógica de recompensas:
            - Movimiento básico: -0.05 (costo de vida)
            - Tocar veneno: -10.0 (penalización fuerte + reset a posición inicial)
            - Recolectar fruta: +1.0 (recompensa por objetivo)
            - Completar nivel: +10.0 (bonus por recolectar todas las frutas)
        """
        # Ejecutar el movimiento según la acción seleccionada
        if action == 0:
            self.agent_pos[0] -= 1    # Mover hacia arriba
        elif action == 1:
            self.agent_pos[0] += 1    # Mover hacia abajo
        elif action == 2:
            self.agent_pos[1] -= 1    # Mover hacia la izquierda
        elif action == 3:
            self.agent_pos[1] += 1    # Mover hacia la derecha

        # Asegurar que el agente permanezca dentro de los límites de la grilla
        self.agent_pos = np.clip(self.agent_pos, 0, self.size - 1)

        # Recompensa base por cada movimiento (costo de vida)
        reward = -0.05
        done = False

        # --- LÓGICA DE MANEJO DE VENENOS ---
        # Verificar si el agente tocó algún veneno
```

```
        if any(np.array_equal(self.agent_pos, p) for p in self.poison_pos):
            reward = -10.0  # Penalización severa por tocar veneno
            self.agent_pos = np.copy(self.start_pos)  # Resetear a posición inicial
            # IMPORTANTE: done NO es True. El episodio continúa después del reset.
        else:
            # --- LÓGICA DE RECOLECCIÓN DE FRUTAS ---
            # Esta lógica solo se ejecuta si NO se tocó un veneno
            eaten_fruit_this_step = False

            # Verificar si el agente recolectó alguna fruta
            for i, fruit in enumerate(self.fruit_pos):
                if np.array_equal(self.agent_pos, fruit):
                    reward += 1.0  # Recompensa por recolectar fruta
                    self.fruit_pos.pop(i)  # Remover la fruta recolectada
                    eaten_fruit_this_step = True
                    break

            # Opcional: Aquí se puede agregar reward shaping basado en distancia
            if not eaten_fruit_this_step and self.fruit_pos:
                # Ejemplo: reward += -0.01 * distancia_a_fruta_más_cercana
                pass  # Actualmente no implementado

            # --- CONDICIÓN DE VICTORIA ---
            # Si no quedan frutas, el episodio termina exitosamente
            if not self.fruit_pos:
                done = True
                reward += 10.0  # Bonus por completar el nivel

        return self.get_state(), reward, done
```

interfaz.py

```
# ddqn_agente_comefrutas.py (versión integrada con interfaz gráfica completa)
"""
Interfaz gráfica para el entrenamiento y visualización de agentes DDQN.

Este módulo implementa una interfaz gráfica completa usando Pygame que permite:
- Configurar entornos de manera interactiva
- Visualizar el comportamiento del agente entrenado
- Alternar entre modo setup y modo juego
- Gestionar elementos del entorno (frutas, venenos, paredes)
```

```
El sistema está diseñado para facilitar la experimentación con diferentes
configuraciones de entorno y la evaluación visual del rendimiento del agente.
"""

import pygame
import numpy as np
import os
import time
import torch
from agent import Agent

# --- CONFIGURACIÓN GENERAL ---
"""Constantes de configuración para la interfaz gráfica."""
GRID_WIDTH = 5          # Ancho de la grilla en celdas
GRID_HEIGHT = 5         # Alto de la grilla en celdas
CELL_SIZE = 120         # Tamaño de cada celda en píxeles
SCREEN_WIDTH = GRID_WIDTH * CELL_SIZE    # Ancho total de la pantalla
SCREEN_HEIGHT = GRID_HEIGHT * CELL_SIZE  # Alto total de la pantalla

# Paleta de colores para la interfaz
COLOR_FONDO = (25, 25, 25)        # Fondo oscuro
COLOR_LINEAS = (40, 40, 40)       # Líneas de la grilla
COLOR_CURSOR = (255, 255, 0)      # Cursor amarillo en modo setup
COLOR_TEXTO = (230, 230, 230)     # Texto claro


# --- ENTORNO PARA DDQN (misma estructura visual que Q-learning) ---
class EntornoGrid:
    """
    Entorno de grilla con interfaz gráfica para agentes DDQN.

    Esta clase maneja tanto la lógica del entorno como su representación visual,
    proporcionando una interfaz interactiva para configurar y visualizar el
    comportamiento del agente. Compatible con la arquitectura DDQN.

    Attributes:
        size (int): Tamaño de la grilla
        agent_pos (tuple): Posición actual del agente (x, y)
        frutas (set): Conjunto de posiciones con frutas
        venenos (set): Conjunto de posiciones con venenos
        paredes (set): Conjunto de posiciones con paredes (obstáculos)
    """
```

```
    def __init__(self):
        """
        Inicializa el entorno de grilla con configuración por defecto.

        El agente comienza en la posición (0,0) y todos los conjuntos de
        elementos están vacíos inicialmente.
        """
        self.size = GRID_WIDTH
        self.agent_pos = (0, 0)
        self.frutas = set()
        self.venenos = set()
        self.paredes = set()

    def reset_a_configuracion_inicial(self):
        """
        Resetea el agente a su posición inicial sin modificar el entorno.

        Esta función es útil para reiniciar episodios manteniendo la misma
        configuración de frutas, venenos y paredes establecida en modo setup.

        Returns:
            np.array: Estado inicial del entorno después del reset.
        """
        self.agent_pos = (0, 0)
        return self.get_state()

    def limpiar_entorno(self):
        """
        Elimina todos los elementos del entorno excepto el agente.

        Esta función es útil para limpiar completamente el entorno y comenzar
        una nueva configuración desde cero en modo setup.
        """
        self.frutas.clear()
        self.venenos.clear()
        self.paredes.clear()

    def step(self, accion):
        """
        Ejecuta una acción del agente y actualiza el estado del entorno.

        Este método implementa la lógica de movimiento y las reglas del juego,
```

```
        incluyendo colisiones con paredes, recolección de frutas y penalizaciones
        por venenos.

        Args:
            accion (int): Acción a ejecutar
                - 0: Mover hacia arriba (y-1)
                - 1: Mover hacia abajo (y+1)
                - 2: Mover hacia la izquierda (x-1)
                - 3: Mover hacia la derecha (x+1)

        Returns:
            tuple: (estado, recompensa, terminado)
                - estado (np.array): Nuevo estado del entorno
                - recompensa (float): Recompensa obtenida por la acción
                - terminado (bool): True si el episodio ha terminado

        Lógica de recompensas:
            - Colisión con pared/límite: -0.1 (sin movimiento)
            - Movimiento válido: -0.05 (costo de vida)
            - Tocar veneno: -10.0 (penalización + reset a origen)
            - Recolectar fruta: +1.0
            - Completar nivel: +10.0 adicional
        """
        x, y = self.agent_pos

        # Calcular nueva posición según la acción
        if accion == 0:
            y -= 1    # Mover hacia arriba
        elif accion == 1:
            y += 1    # Mover hacia abajo
        elif accion == 2:
            x -= 1    # Mover hacia la izquierda
        elif accion == 3:
            x += 1    # Mover hacia la derecha

        # Verificar colisiones con límites de la grilla o paredes
        if (
            x < 0
            or x >= GRID_WIDTH
            or y < 0
            or y >= GRID_HEIGHT
            or (x, y) in self.paredes
```

```
        ):
            # Movimiento inválido: penalización menor y no se mueve
            return self.get_state(), -0.1, False

        # Movimiento válido: actualizar posición del agente
        self.agent_pos = (x, y)
        recompensa = -0.05  # Costo base por movimiento
        terminado = False

        # Verificar interacciones con elementos del entorno
        if self.agent_pos in self.venenos:
            # Penalización por tocar veneno y reset a posición inicial
            recompensa = -10.0
            self.agent_pos = (0, 0)
        elif self.agent_pos in self.frutas:
            # Recompensa por recolectar fruta
            recompensa = 1.0
            self.frutas.remove(self.agent_pos)

            # Verificar si se completó el nivel (no quedan frutas)
            if not self.frutas:
                recompensa += 10.0  # Bonus por completar
                terminado = True

        return self.get_state(), recompensa, terminado

    def get_state(self):
        """
        Obtiene la representación del estado actual como tensor 3D.

        Convierte el estado del entorno en un formato compatible con redes
        neuronales convolucionales, usando 3 canales para representar
        diferentes tipos de elementos.

        Returns:
            np.array: Tensor 3D de forma (3, size, size) donde:
                - Canal 0: Posición del agente (1.0 donde está el agente)
                - Canal 1: Posiciones de frutas (1.0 donde hay frutas)
                - Canal 2: Posiciones de venenos (1.0 donde hay venenos)

        Nota: Las paredes no se incluyen en el estado ya que son estáticas
              y se manejan a través de las restricciones de movimiento.
```

```
        """
        estado = np.zeros((3, self.size, self.size), dtype=np.float32)

        # Canal 0: Posición del agente
        estado[0, self.agent_pos[0], self.agent_pos[1]] = 1.0

        # Canal 1: Posiciones de frutas
        for fruta in self.frutas:
            estado[1, fruta[0], fruta[1]] = 1.0

        # Canal 2: Posiciones de venenos
        for veneno in self.venenos:
            estado[2, veneno[0], veneno[1]] = 1.0

        return estado

    def dibujar(
        self,
        pantalla,
        modo_juego,
        cursor_pos,
        img_fruta,
        img_veneno,
        img_pared,
        img_agente,
    ):
        """
        Renderiza el entorno completo en la pantalla de Pygame.

        Este método se encarga de dibujar todos los elementos visuales del juego,
        incluyendo la grilla, elementos del entorno, el agente, el cursor (en modo setup)
        y la información de controles.

        Args:
            pantalla (pygame.Surface): Superficie donde dibujar
            modo_juego (str): Modo actual ("SETUP" o "PLAYING")
            cursor_pos (tuple): Posición del cursor en modo setup
            img_fruta (pygame.Surface): Imagen de la fruta
            img_veneno (pygame.Surface): Imagen del veneno
            img_pared (pygame.Surface): Imagen de la pared
            img_agente (pygame.Surface): Imagen del agente
```

```
        Elementos visuales renderizados:
            1. Fondo y grilla
            2. Paredes (obstáculos estáticos)
            3. Frutas (objetivos a recolectar)
            4. Venenos (elementos a evitar)
            5. Agente (jugador controlado por IA)
            6. Cursor (solo en modo setup)
            7. Información de controles y modo actual
        """
        # Limpiar pantalla con color de fondo
        pantalla.fill(COLOR_FONDO)

        # Dibujar líneas de la grilla (verticales)
        for x in range(0, SCREEN_WIDTH, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (x, 0), (x, SCREEN_HEIGHT))

        # Dibujar líneas de la grilla (horizontales)
        for y in range(0, SCREEN_HEIGHT, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (0, y), (SCREEN_WIDTH, y))

        # Dibujar elementos del entorno en orden de capas
        # 1. Paredes (fondo)
        for pared in self.paredes:
            pantalla.blit(img_pared, (pared[0] * CELL_SIZE, pared[1] * CELL_SIZE))

        # 2. Frutas (objetivos)
        for fruta in self.frutas:
            pantalla.blit(img_fruta, (fruta[0] * CELL_SIZE, fruta[1] * CELL_SIZE))

        # 3. Venenos (peligros)
        for veneno in self.venenos:
            pantalla.blit(img_veneno, (veneno[0] * CELL_SIZE, veneno[1] * CELL_SIZE))

        # 4. Agente (primer plano)
        pantalla.blit(
            img_agente, (self.agent_pos[0] * CELL_SIZE, self.agent_pos[1] * CELL_SIZE)
        )

        # 5. Cursor (solo en modo setup)
        if modo_juego == "SETUP":
            cursor_rect = pygame.Rect(
                cursor_pos[0] * CELL_SIZE,
```

```
                cursor_pos[1] * CELL_SIZE,
                CELL_SIZE,
                CELL_SIZE,
            )
            pygame.draw.rect(pantalla, COLOR_CURSOR, cursor_rect, 3)

        # Renderizar información textual
        font = pygame.font.Font(None, 24)

        # Información del modo actual
        texto_modo = font.render(f"Modo: {modo_juego}", True, COLOR_TEXTO)

        # Controles para modo setup
        controles1 = font.render(
            "SETUP: Flechas, F=Fruta, V=Veneno, W=Pared, C=Limpiar", True, COLOR_TEXTO
        )

        # Controles generales
        controles2 = font.render("P=Jugar, S=Setup", True, COLOR_TEXTO)

        # Posicionar textos en la parte inferior
        pantalla.blit(texto_modo, (10, SCREEN_HEIGHT + 5))
        pantalla.blit(controles1, (10, SCREEN_HEIGHT + 30))
        pantalla.blit(controles2, (10, SCREEN_HEIGHT + 55))


# --- MAIN CON INTERFAZ COMPLETA ---
def main():
    """
    Función principal que ejecuta la interfaz gráfica completa del sistema DDQN.

    Esta función implementa el bucle principal del programa, manejando:
    - Inicialización de Pygame y carga de recursos
    - Gestión de eventos de teclado para ambos modos
    - Alternancia entre modo setup y modo juego
    - Renderizado continuo de la interfaz
    - Ejecución automática del agente en modo juego

    Modos de operación:
        SETUP: Permite configurar el entorno interactivamente
            - Flechas: Mover cursor
            - F: Añadir/quitar fruta
```

```
            - V: Añadir/quitar veneno
            - W: Añadir/quitar pared
            - C: Limpiar entorno

        PLAYING: El agente entrenado juega automáticamente
            - Usa el modelo DDQN cargado para tomar decisiones
            - Visualiza el comportamiento del agente en tiempo real
            - Termina automáticamente y vuelve a setup al completar
    """
    # Inicializar Pygame y crear ventana
    pygame.init()
    pantalla = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
    pygame.display.set_caption("Agente DDQN - Come Frutas  ")

    def cargar_img(nombre, color_fallback):
        """
        Carga una imagen desde archivo con fallback a color sólido.

        Esta función auxiliar intenta cargar una imagen desde el directorio
        del script. Si falla, crea una superficie de color sólido como respaldo.

        Args:
            nombre (str): Nombre del archivo de imagen
            color_fallback (tuple): Color RGB de respaldo si falla la carga

        Returns:
            pygame.Surface: Superficie escalada al tamaño de celda
        """
        try:
            ruta = os.path.join(os.path.dirname(__file__), nombre)
            img = pygame.image.load(ruta).convert_alpha()
            return pygame.transform.scale(img, (CELL_SIZE, CELL_SIZE))
        except:
            # Crear superficie de color sólido como fallback
            surf = pygame.Surface((CELL_SIZE, CELL_SIZE))
            surf.fill(color_fallback)
            return surf

    # Cargar imágenes con colores de respaldo
    img_fruta = cargar_img("fruta.png", (0, 255, 0))      # Verde si falla
    img_veneno = cargar_img("veneno.png", (255, 0, 0))     # Rojo si falla
    img_pared = cargar_img("pared.png", (100, 100, 100))   # Gris si falla
```

```
    img_agente = cargar_img("agente.png", (0, 0, 255))     # Azul si falla

    # Inicializar componentes principales
    entorno = EntornoGrid()
    agente = Agent(state_shape=(3, GRID_HEIGHT, GRID_WIDTH), action_size=4)
    agente.load("dqn_model.pth")  # Cargar modelo entrenado

    # Variables de control de la interfaz
    cursor_pos = [0, 0]      # Posición del cursor en modo setup
    modo_juego = "SETUP"     # Modo inicial
    reloj = pygame.time.Clock()  # Control de FPS
    corriendo = True

    # Bucle principal del programa
    while corriendo:
        # Procesar eventos de Pygame
        for evento in pygame.event.get():
            if evento.type == pygame.QUIT:
                corriendo = False

            if evento.type == pygame.KEYDOWN:
                # Cambios de modo (disponibles en cualquier momento)
                if evento.key == pygame.K_p:
                    print("--- MODO JUEGO ---")
                    entorno.reset_a_configuracion_inicial()
                    modo_juego = "PLAYING"
                    time.sleep(0.5)  # Pausa para evitar acciones inmediatas

                elif evento.key == pygame.K_s:
                    print("--- MODO SETUP ---")
                    modo_juego = "SETUP"

                # Controles específicos del modo SETUP
                if modo_juego == "SETUP":
                    # Movimiento del cursor
                    if evento.key == pygame.K_UP:
                        cursor_pos[1] = max(0, cursor_pos[1] - 1)
                    elif evento.key == pygame.K_DOWN:
                        cursor_pos[1] = min(GRID_HEIGHT - 1, cursor_pos[1] + 1)
                    elif evento.key == pygame.K_LEFT:
                        cursor_pos[0] = max(0, cursor_pos[0] - 1)
                    elif evento.key == pygame.K_RIGHT:
```

```
                    cursor_pos[0] = min(GRID_WIDTH - 1, cursor_pos[0] + 1)

                # Gestión de elementos en la posición del cursor
                pos = tuple(cursor_pos)

                if evento.key == pygame.K_f:
                    # Alternar fruta en posición actual
                    if pos in entorno.frutas:
                        entorno.frutas.remove(pos)
                    else:
                        entorno.frutas.add(pos)
                        # Remover otros elementos de la misma posición
                        entorno.venenos.discard(pos)
                        entorno.paredes.discard(pos)

                elif evento.key == pygame.K_v:
                    # Alternar veneno en posición actual
                    if pos in entorno.venenos:
                        entorno.venenos.remove(pos)
                    else:
                        entorno.venenos.add(pos)
                        # Remover otros elementos de la misma posición
                        entorno.frutas.discard(pos)
                        entorno.paredes.discard(pos)

                elif evento.key == pygame.K_w:
                    # Alternar pared en posición actual
                    if pos in entorno.paredes:
                        entorno.paredes.remove(pos)
                    else:
                        entorno.paredes.add(pos)
                        # Remover otros elementos de la misma posición
                        entorno.frutas.discard(pos)
                        entorno.venenos.discard(pos)

                elif evento.key == pygame.K_c:
                    # Limpiar todo el entorno
                    print("--- LIMPIANDO ENTORNO ---")
                    entorno.limpiar_entorno()

        # Lógica del modo PLAYING (agente automático)
        if modo_juego == "PLAYING":
```

```
            # Obtener estado actual del entorno
            estado = entorno.get_state()

            # El agente elige una acción usando el modelo entrenado
            # explore=False significa que usa solo explotación (sin exploración)
            accion = agente.choose_action(estado, explore=False)

            # Ejecutar la acción en el entorno
            _, _, terminado = entorno.step(accion)

            # Si el episodio terminó, volver al modo setup
            if terminado:
                print("Juego terminado. Volviendo a SETUP.")
                modo_juego = "SETUP"

            # Controlar velocidad de visualización (10 FPS para el agente)
            time.sleep(0.1)

        # Renderizado de la interfaz
        # Crear superficie completa incluyendo espacio para texto
        pantalla_con_info = pygame.Surface((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
        pantalla_con_info.fill(COLOR_FONDO)

        # Dibujar el entorno en la superficie
        entorno.dibujar(
            pantalla_con_info,
            modo_juego,
            tuple(cursor_pos),
            img_fruta,
            img_veneno,
            img_pared,
            img_agente,
        )

        # Copiar la superficie completa a la pantalla principal
        pantalla.blit(pantalla_con_info, (0, 0))

        # Actualizar la pantalla y controlar FPS
        pygame.display.flip()
        reloj.tick(60)  # Limitar a 60 FPS para suavidad visual

    # Limpiar recursos al salir
```

```
    pygame.quit()


if __name__ == "__main__":
    """
    Punto de entrada del programa.

    Ejecuta la función main() solo cuando el archivo se ejecuta directamente,
    no cuando se importa como módulo. Esto permite reutilizar las clases
    y funciones en otros scripts sin ejecutar automáticamente la interfaz.
    """
    main()
```

interfaztrain.py

```
# ddqn_agente_comefrutas.py
"""
Interfaz de entrenamiento interactiva para agentes DDQN.

Este módulo proporciona una interfaz gráfica completa que permite:
- Configurar entornos de entrenamiento de manera interactiva
- Entrenar agentes DDQN con visualización en tiempo real
- Evaluar el rendimiento del agente después del entrenamiento
- Gestionar el ciclo completo de desarrollo de IA: setup → entrenamiento → evaluación

La interfaz integra tres modos principales:
1. SETUP: Configuración interactiva del entorno
2. TRAINING: Entrenamiento automático del agente DDQN
3. PLAYING: Evaluación visual del agente entrenado

Diseñado para facilitar la experimentación y el desarrollo iterativo de agentes
de aprendizaje por refuerzo en entornos de grilla.
"""

import pygame
import numpy as np
import os
import time
from agent import Agent
from environment import GridEnvironment
```

```
# --- CONFIGURACIÓN DEL ENTORNO Y VISUALIZACIÓN ---
"""Parámetros principales del sistema de entrenamiento."""
GRID_SIZE = 5           # Tamaño de la grilla (5x5)
CELL_SIZE = 120         # Tamaño en píxeles de cada celda
SCREEN_WIDTH = GRID_SIZE * CELL_SIZE    # Ancho total de la ventana
SCREEN_HEIGHT = GRID_SIZE * CELL_SIZE   # Alto total de la ventana

# Esquema de colores para la interfaz
COLOR_FONDO = (25, 25, 25)        # Fondo oscuro para mejor contraste
COLOR_LINEAS = (40, 40, 40)       # Líneas sutiles de la grilla
COLOR_CURSOR = (255, 255, 0)      # Cursor amarillo brillante
COLOR_TEXTO = (230, 230, 230)     # Texto claro y legible

# --- PARÁMETROS DE ENTRENAMIENTO ---
"""Configuración del proceso de entrenamiento DDQN."""
NUM_EPISODIOS_ENTRENAMIENTO = 3000    # Número total de episodios de entrenamiento
BATCH_SIZE = 128                      # Tamaño del lote para replay de experiencias


def cargar_imagen(ruta, color_si_falla):
    """
    Carga una imagen desde archivo con sistema de fallback robusto.

    Esta función implementa un mecanismo de carga de imágenes que garantiza
    que el programa funcione incluso si los archivos de imagen no están
    disponibles, creando superficies de color como respaldo.

    Args:
        ruta (str): Ruta relativa o absoluta al archivo de imagen
        color_si_falla (tuple): Color RGB (r, g, b) a usar si falla la carga

    Returns:
        pygame.Surface: Superficie escalada al tamaño de celda, ya sea la
                        imagen cargada o una superficie de color sólido

    Características:
        - Manejo automático de transparencia (convert_alpha)
        - Escalado automático al tamaño de celda
        - Fallback graceful a color sólido
        - Compatible con todos los formatos soportados por Pygame
    """
    try:
```

```
        # Intentar cargar la imagen desde archivo
        img = pygame.image.load(ruta).convert_alpha()
        # Escalar al tamaño exacto de celda para consistencia visual
        return pygame.transform.scale(img, (CELL_SIZE, CELL_SIZE))
    except pygame.error:
        # Si falla la carga, crear superficie de color sólido
        surf = pygame.Surface((CELL_SIZE, CELL_SIZE))
        surf.fill(color_si_falla)
        return surf


def main():
    """
    Función principal que ejecuta la interfaz de entrenamiento DDQN.

    Esta función implementa un sistema completo de desarrollo de agentes IA que incluye:

    1. **Configuración Interactiva (Modo SETUP)**:
       - Diseño visual del entorno usando cursor
       - Colocación de frutas, venenos y paredes
       - Validación de configuraciones

    2. **Entrenamiento Automatizado (Modo TRAINING)**:
       - Ejecución de algoritmo DDQN completo
       - Actualización de redes objetivo
       - Monitoreo de progreso en tiempo real
       - Gestión de memoria de experiencias

    3. **Evaluación Visual (Modo PLAYING)**:
       - Visualización del comportamiento aprendido
       - Modo sin exploración (solo explotación)
       - Análisis cualitativo del rendimiento

    Flujo de trabajo típico:
        SETUP → TRAINING → PLAYING → [iteración]

    La interfaz permite experimentación rápida con diferentes configuraciones
    de entorno y hiperparámetros de entrenamiento.
    """
    # Inicialización del sistema gráfico
    pygame.init()
    pantalla = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
```

```
    pygame.display.set_caption("Agente Come-Frutas DDQN  ")

    # Cargar recursos visuales con colores de fallback específicos
    img_fruta = cargar_imagen("fruta.png", (40, 200, 40))    # Verde si falla
    img_veneno = cargar_imagen("veneno.png", (200, 40, 40))   # Rojo si falla
    img_pared = cargar_imagen("pared.png", (100, 100, 100))   # Gris si falla
    img_agente = cargar_imagen("agente.png", (40, 200, 40))   # Verde si falla

    # Inicialización de componentes principales del sistema
    entorno = GridEnvironment(size=GRID_SIZE)  # Entorno de simulación
    agente = Agent(state_shape=(3, GRID_SIZE, GRID_SIZE), action_size=4)  # Agente DDQN

    # Variables de control de la interfaz
    cursor_pos = [0, 0]        # Posición del cursor en modo setup
    modo_juego = "SETUP"       # Estado inicial del sistema
    reloj = pygame.time.Clock()  # Control de framerate
    corriendo = True           # Flag principal del bucle

    # Conjuntos para gestionar elementos del entorno configurables
    frutas = set()    # Posiciones de objetivos (recompensa positiva)
    venenos = set()   # Posiciones de peligros (penalización)
    paredes = set()   # Posiciones de obstáculos (bloqueo de movimiento)

    # Bucle principal del sistema de entrenamiento
    while corriendo:
        # Procesamiento de eventos del usuario
        for evento in pygame.event.get():
            if evento.type == pygame.QUIT:
                corriendo = False

            if evento.type == pygame.KEYDOWN:
                # --- CONTROL DE ENTRENAMIENTO (Tecla T) ---
                if evento.key == pygame.K_t:
                    if modo_juego != "TRAINING":
                        print("--- ENTRENANDO DDQN ---")
                        modo_juego = "TRAINING"

                        # Bucle principal de entrenamiento DDQN
                        for episodio in range(NUM_EPISODIOS_ENTRENAMIENTO):
                            # Reiniciar entorno con configuración actual
                            estado = entorno.reset(
                                agent_pos=(0, 0),
```

```
                            fruit_pos=list(frutas),
                            poison_pos=list(venenos),
                        )

                        # Variables de control del episodio
                        terminado = False
                        total_reward = 0

                        # Bucle del episodio individual
                        while not terminado:
                            # El agente elige acción con exploración activa
                            accion = agente.choose_action(estado, explore=True)

                            # Ejecutar acción y observar resultado
                            nuevo_estado, recompensa, terminado = entorno.step(accion)

                            # Almacenar experiencia en memoria de replay
                            agente.remember(
                                estado, accion, recompensa, nuevo_estado, terminado
                            )

                            # Entrenar la red con experiencias pasadas
                            agente.replay(BATCH_SIZE)

                            # Actualización periódica de la red objetivo
                            if agente.steps_done % agente.update_target_every == 0:
                                agente.update_target_network()

                            # Preparar para siguiente paso
                            estado = nuevo_estado
                            total_reward += recompensa

                        # Reporte de progreso cada 100 episodios
                        if (episodio + 1) % 100 == 0:
                            print(
                                f"Ep {episodio+1}, Reward: {total_reward:.2f}, Epsilon:
                            )

                    print("--- ENTRENAMIENTO COMPLETO ---")
                    modo_juego = "PLAYING"  # Cambiar automáticamente a evaluación

                # --- CONTROL DE MODOS (Teclas P y S) ---
```

```
                elif evento.key == pygame.K_p:
                    print("--- MODO PLAYING ---")
                    # Reiniciar entorno para evaluación
                    entorno.reset(
                        agent_pos=(0, 0),
                        fruit_pos=list(frutas),
                        poison_pos=list(venenos),
                    )
                    modo_juego = "PLAYING"

                elif evento.key == pygame.K_s:
                    print("--- MODO SETUP ---")
                    modo_juego = "SETUP"

                # --- CONTROLES DEL MODO SETUP ---
                if modo_juego == "SETUP":
                    # Control de navegación del cursor
                    if evento.key == pygame.K_UP:
                        cursor_pos[1] = max(0, cursor_pos[1] - 1)
                    elif evento.key == pygame.K_DOWN:
                        cursor_pos[1] = min(GRID_SIZE - 1, cursor_pos[1] + 1)
                    elif evento.key == pygame.K_LEFT:
                        cursor_pos[0] = max(0, cursor_pos[0] - 1)
                    elif evento.key == pygame.K_RIGHT:
                        cursor_pos[0] = min(GRID_SIZE - 1, cursor_pos[0] + 1)

                    # Conversión de coordenadas (cursor usa x,y pero entorno usa y,x)
                    pos = tuple(cursor_pos[::-1])

                    # Gestión de elementos en la posición del cursor
                    if evento.key == pygame.K_f:
                        # Alternar fruta en posición actual
                        if pos in frutas:
                            frutas.remove(pos)
                        else:
                            frutas.add(pos)
                            # Limpiar otros elementos de la misma posición
                            venenos.discard(pos)
                            paredes.discard(pos)

                    elif evento.key == pygame.K_v:
                        # Alternar veneno en posición actual
```

```
                    if pos in venenos:
                        venenos.remove(pos)
                    else:
                        venenos.add(pos)
                        # Limpiar otros elementos de la misma posición
                        frutas.discard(pos)
                        paredes.discard(pos)

                elif evento.key == pygame.K_w:
                    # Alternar pared en posición actual
                    if pos in paredes:
                        paredes.remove(pos)
                    else:
                        paredes.add(pos)
                        # Limpiar otros elementos de la misma posición
                        frutas.discard(pos)
                        venenos.discard(pos)

                elif evento.key == pygame.K_c:
                    # Limpiar completamente el entorno
                    frutas.clear()
                    venenos.clear()
                    paredes.clear()

    # --- LÓGICA DEL MODO PLAYING ---
    if modo_juego == "PLAYING":
        # Obtener estado actual del entorno
        estado = entorno.get_state()

        # El agente toma decisiones sin exploración (solo explotación)
        accion = agente.choose_action(estado, explore=False)

        # Ejecutar acción y verificar si terminó el episodio
        _, _, terminado = entorno.step(accion)

        if terminado:
            print("Juego terminado. Volviendo a SETUP.")
            modo_juego = "SETUP"

        # Control de velocidad de visualización
        time.sleep(0.1)  # 10 FPS para observar mejor el comportamiento
```

```
        # --- SISTEMA DE RENDERIZADO COMPLETO ---
        # Limpiar pantalla con color de fondo
        pantalla.fill(COLOR_FONDO)

        # Dibujar grilla de referencia
        for x in range(0, SCREEN_WIDTH, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (x, 0), (x, SCREEN_HEIGHT))
        for y in range(0, SCREEN_HEIGHT, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (0, y), (SCREEN_WIDTH, y))

        # Renderizar elementos del entorno (orden de capas importante)
        # Nota: Las coordenadas se invierten para coincidir con el sistema visual

        # 1. Paredes (capa de fondo)
        for pared in paredes:
            pantalla.blit(img_pared, (pared[1] * CELL_SIZE, pared[0] * CELL_SIZE))

        # 2. Frutas (objetivos)
        for fruta in frutas:
            pantalla.blit(img_fruta, (fruta[1] * CELL_SIZE, fruta[0] * CELL_SIZE))

        # 3. Venenos (peligros)
        for veneno in venenos:
            pantalla.blit(img_veneno, (veneno[1] * CELL_SIZE, veneno[0] * CELL_SIZE))

        # 4. Agente (solo visible fuera del modo setup)
        if modo_juego != "SETUP":
            pos = entorno.agent_pos
            pantalla.blit(img_agente, (pos[1] * CELL_SIZE, pos[0] * CELL_SIZE))

        # 5. Cursor (solo visible en modo setup)
        if modo_juego == "SETUP":
            cursor_rect = pygame.Rect(
                cursor_pos[0] * CELL_SIZE,
                cursor_pos[1] * CELL_SIZE,
                CELL_SIZE,
                CELL_SIZE,
            )
            pygame.draw.rect(pantalla, COLOR_CURSOR, cursor_rect, 3)

        # --- INTERFAZ DE INFORMACIÓN Y CONTROLES ---
        font = pygame.font.Font(None, 24)
```

```
        # Mostrar modo actual
        pantalla.blit(
            font.render(f"Modo: {modo_juego}", True, COLOR_TEXTO),
            (10, SCREEN_HEIGHT + 5),
        )

        # Controles para modo setup
        pantalla.blit(
            font.render(
                "SETUP: Flechas, F=Fruta, V=Veneno, W=Pared, C=Limpiar",
                True,
                COLOR_TEXTO,
            ),
            (10, SCREEN_HEIGHT + 30),
        )

        # Controles generales del sistema
        pantalla.blit(
            font.render("T=Entrenar, P=Jugar, S=Setup", True, COLOR_TEXTO),
            (10, SCREEN_HEIGHT + 55),
        )

        # Actualizar pantalla y controlar framerate
        pygame.display.flip()
        reloj.tick(60)  # Limitar a 60 FPS para suavidad visual

    # Limpiar recursos al finalizar
    pygame.quit()


if __name__ == "__main__":
    """
    Punto de entrada del programa de entrenamiento.

    Ejecuta la función main() cuando el archivo se ejecuta directamente.
    Este patrón permite importar las funciones y clases de este módulo
    en otros scripts sin ejecutar automáticamente la interfaz de entrenamiento.

    Uso típico:
        python interfaztrain.py  # Ejecuta la interfaz completa

    O desde otro script:
```

```
        from interfaztrain import cargar_imagen  # Solo importa funciones
    """
    main()
```

train.py

```
# train.py
"""
Script de entrenamiento principal para el agente DDQN (Double Deep Q-Network).

Este módulo implementa el proceso completo de entrenamiento del agente DDQN para
el problema de recolección de frutas evitando venenos. El entrenamiento utiliza
generación aleatoria de escenarios para garantizar la generalización del agente.

Características principales del entrenamiento:
- Generación aleatoria de entornos para cada episodio
- Implementación completa del algoritmo DDQN
- Actualización periódica de la red objetivo
- Guardado automático del modelo durante el entrenamiento
- Monitoreo del progreso con métricas de rendimiento

El sistema está diseñado para entrenar un agente robusto capaz de manejar
una amplia variedad de configuraciones de entorno, desde escenarios simples
hasta configuraciones complejas con múltiples obstáculos y objetivos.

Algoritmo implementado:
- Double Deep Q-Network (DDQN) con replay buffer
- Exploración epsilon-greedy con decaimiento
- Actualización periódica de red objetivo
- Entrenamiento continuo con experiencias almacenadas
"""

from environment import GridEnvironment
from agent import Agent
import numpy as np
import random

# --- CONFIGURACIÓN DE ENTRENAMIENTO ---
"""Hiperparámetros principales del proceso de entrenamiento."""
EPISODES = 25000    # Número total de episodios de entrenamiento (juegos completos)
GRID_SIZE = 5       # Tamaño de la grilla del entorno (5x5 celdas)
```

```python
if __name__ == "__main__":
    """
    Función principal que ejecuta el proceso completo de entrenamiento DDQN.

    Este bloque implementa el algoritmo de entrenamiento completo, incluyendo:
    - Inicialización del entorno y agente
    - Generación aleatoria de escenarios de entrenamiento
    - Bucle principal de entrenamiento con DDQN
    - Gestión de experiencias y actualización de redes
    - Monitoreo y guardado del progreso

    El entrenamiento utiliza curriculum learning implícito a través de la
    variabilidad aleatoria de escenarios, exponiendo al agente a una amplia
    gama de situaciones para mejorar la generalización.
    """

    # --- INICIALIZACIÓN DE COMPONENTES ---
    env = GridEnvironment(size=GRID_SIZE)              # Entorno de simulación
    state_shape = (3, GRID_SIZE, GRID_SIZE)           # Forma del estado: 3 canales x 5x5
    action_size = 4                                    # Número de acciones posibles (4 direc
    agent = Agent(state_shape, action_size)           # Agente DDQN con arquitectura CNN

    # --- CONFIGURACIÓN DE HIPERPARÁMETROS ---
    batch_size = 128    # Tamaño del lote para entrenamiento de la red neural
                        # Un batch size mayor proporciona gradientes más estables

    # --- BUCLE PRINCIPAL DE ENTRENAMIENTO ---
    for e in range(EPISODES):
        # --- GENERACIÓN ALEATORIA DE ESCENARIOS ---
        """
        Cada episodio utiliza una configuración completamente aleatoria del entorno.
        Esta estrategia es FUNDAMENTAL para la generalización del agente, ya que
        evita el sobreajuste a configuraciones específicas y fuerza al agente
        a aprender estrategias robustas que funcionen en cualquier escenario.
        """

        # Determinar número aleatorio de elementos en el entorno
        num_fruits = np.random.randint(1, 5)    # Entre 1 y 4 frutas
        num_poisons = np.random.randint(1, 4)   # Entre 1 y 3 venenos

        # Generar posiciones únicas para todos los elementos
        # Esto previene superposiciones y garantiza configuraciones válidas
```

```
        all_pos = [(i, j) for i in range(GRID_SIZE) for j in range(GRID_SIZE)]
        random.shuffle(all_pos)  # Mezclar aleatoriamente todas las posiciones

        # Asignar posiciones únicas para cada elemento
        agent_pos = all_pos.pop()                           # Posición inicial del agente
        fruit_pos = [all_pos.pop() for _ in range(num_fruits)]   # Posiciones de frutas
        poison_pos = [all_pos.pop() for _ in range(num_poisons)] # Posiciones de venenos

        # Reiniciar entorno con la configuración generada
        state = env.reset(agent_pos=agent_pos, fruit_pos=fruit_pos, poison_pos=poison_pos)

        # --- EJECUCIÓN DEL EPISODIO ---
        """
        Cada episodio simula un juego completo donde el agente debe recolectar
        todas las frutas mientras evita los venenos. El límite de 50 pasos
        previene episodios infinitos y fuerza al agente a ser eficiente.
        """
        total_reward = 0

        # Bucle de pasos dentro del episodio (máximo 50 pasos)
        for time in range(50):
            # El agente elige una acción usando la política epsilon-greedy
            # Durante el entrenamiento, explora aleatoriamente con probabilidad epsilon
            action = agent.choose_action(state)

            # Ejecutar la acción en el entorno
            next_state, reward, done = env.step(action)

            # Almacenar la experiencia en el buffer de replay
            # Esta experiencia se usará más tarde para entrenar la red
            agent.remember(state, action, reward, next_state, done)

            # Actualizar estado y acumular recompensa
            state = next_state
            total_reward += reward

            # --- ACTUALIZACIÓN DE LA RED OBJETIVO ---
            """
            La red objetivo se actualiza periódicamente para estabilizar el entrenamiento.
            Esto es una característica clave del algoritmo DQN que previene la
            divergencia durante el entrenamiento.
            """
```

```
            if agent.steps_done % agent.update_target_every == 0:
                agent.update_target_network()

            # Terminar episodio si se completó el objetivo
            if done:
                break

        # --- MONITOREO DEL PROGRESO ---
        """
        Imprimir estadísticas del episodio para monitorear el progreso del entrenamiento.
        - Puntuación total: Indica qué tan bien está aprendiendo el agente
        - Epsilon: Muestra el balance actual entre exploración y explotación
        """
        print(f"Episodio: {e+1}/{EPISODES}, Puntuación: {total_reward:.2f}, Epsilon: {agent.e

        # --- ENTRENAMIENTO DE LA RED NEURAL ---
        """
        El entrenamiento se realiza después de cada episodio usando experiencias
        almacenadas en el buffer de replay. Esto permite que el agente aprenda
        de experiencias pasadas, mejorando la eficiencia del aprendizaje.
        """
        agent.replay(batch_size)

        # --- GUARDADO PERIÓDICO DEL MODELO ---
        """
        Guardar el modelo cada 50 episodios para:
        - Prevenir pérdida de progreso en caso de interrupciones
        - Permitir evaluación de versiones intermedias
        - Facilitar la reanudación del entrenamiento si es necesario
        """
        if e % 50 == 0:
            agent.save("dqn_model.pth")

    # --- FINALIZACIÓN DEL ENTRENAMIENTO ---
    print("Entrenamiento finalizado. Modelo guardado en 'dqn_model.pth'")
```

main.py

```
# main.py
"""
Demostración interactiva del agente DDQN entrenado.
```

```
Este módulo proporciona una interfaz de demostración simple donde los usuarios pueden:
- Configurar entornos personalizados mediante clics del mouse
- Observar el comportamiento del agente DDQN entrenado en tiempo real
- Experimentar con diferentes configuraciones de frutas y venenos

El sistema está diseñado como una demostración pública o para validación
del rendimiento del agente en escenarios definidos por el usuario.

Características principales:
- Interfaz minimalista y fácil de usar
- Carga automática del modelo entrenado
- Visualización en tiempo real del agente
- Reinicio automático para múltiples demostraciones

Flujo de trabajo:
1. Modo SETUP: El usuario configura frutas y venenos con clics
2. Modo RUN: El agente ejecuta la solución automáticamente
3. Reinicio automático al completar la demostración
"""

import pygame
import numpy as np
from environment import GridEnvironment
from agent import Agent

# --- CONFIGURACIÓN DE LA INTERFAZ GRÁFICA ---
"""Parámetros de visualización y configuración de la ventana."""
GRID_SIZE = 5           # Tamaño de la grilla (5x5)
CELL_SIZE = 100         # Tamaño de cada celda en píxeles
WIDTH, HEIGHT = GRID_SIZE * CELL_SIZE, GRID_SIZE * CELL_SIZE  # Dimensiones de ventana
WIN = pygame.display.set_mode((WIDTH, HEIGHT))               # Ventana principal
pygame.display.set_caption("Agente Come-Frutas")            # Título de la ventana
pygame.font.init()                                           # Inicializar sistema de fuentes

# --- INICIALIZACIÓN DEL AGENTE ENTRENADO ---
"""
Configuración y carga del agente DDQN preentrenado.

Este bloque inicializa los componentes necesarios para ejecutar
el agente entrenado en modo demostración.
"""
env = GridEnvironment(size=GRID_SIZE)                    # Entorno de simulación
```

```
action_size = 4                                          # Número de acciones posibles (4 di
state_shape = (3, GRID_SIZE, GRID_SIZE)                  # Forma del estado: 3 canales x 5x5
agent = Agent(state_shape, action_size)                  # Crear instancia del agente
agent.load("dqn_model.pth")                             # Cargar modelo preentrenado

# --- ESQUEMA DE COLORES PARA VISUALIZACIÓN ---
"""Colores RGB para los diferentes elementos del juego."""
COLOR_GRID = (200, 200, 200)    # Gris claro para las líneas de la grilla
COLOR_AGENT = (0, 0, 255)       # Azul para el agente
COLOR_FRUIT = (0, 255, 0)       # Verde para las frutas (objetivos)
COLOR_POISON = (255, 0, 0)      # Rojo para los venenos (peligros)


def draw_grid():
    """
    Dibuja las líneas de la grilla en la ventana.

    Crea una cuadrícula visual que ayuda a los usuarios a identificar
    las posiciones disponibles para colocar elementos durante el modo setup.

    La grilla se dibuja con líneas verticales y horizontales espaciadas
    uniformemente según el tamaño de celda configurado.
    """
    # Dibujar líneas verticales
    for x in range(0, WIDTH, CELL_SIZE):
        pygame.draw.line(WIN, COLOR_GRID, (x, 0), (x, HEIGHT))

    # Dibujar líneas horizontales
    for y in range(0, HEIGHT, CELL_SIZE):
        pygame.draw.line(WIN, COLOR_GRID, (0, y), (WIDTH, y))


def draw_elements(agent_pos, fruits, poisons):
    """
    Renderiza todos los elementos del juego en la pantalla.

    Esta función se encarga de dibujar visualmente todos los componentes
    del entorno: el agente, las frutas y los venenos, usando formas
    geométricas distintivas para cada tipo de elemento.

    Args:
        agent_pos (np.array): Posición actual del agente [fila, columna]
```

```
        fruits (list): Lista de posiciones de frutas [(fila, col), ...]
        poisons (list): Lista de posiciones de venenos [(fila, col), ...]

    Representaciones visuales:
        - Agente: Rectángulo azul que ocupa toda la celda
        - Frutas: Círculos verdes centrados en las celdas
        - Venenos: Cuadrados rojos más pequeños dentro de las celdas
    """
    # Dibujar agente como rectángulo azul
    # Nota: Se intercambian coordenadas (agent_pos[1], agent_pos[0]) para
    # convertir de coordenadas de matriz (fila, columna) a pantalla (x, y)
    pygame.draw.rect(
        WIN,
        COLOR_AGENT,
        (agent_pos[1] * CELL_SIZE, agent_pos[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE),
    )

    # Dibujar frutas como círculos verdes
    for f in fruits:
        center_x = f[1] * CELL_SIZE + CELL_SIZE // 2  # Centro horizontal
        center_y = f[0] * CELL_SIZE + CELL_SIZE // 2  # Centro vertical
        radius = CELL_SIZE // 3                       # Radio del círculo
        pygame.draw.circle(WIN, COLOR_FRUIT, (center_x, center_y), radius)

    # Dibujar venenos como cuadrados rojos más pequeños
    for p in poisons:
        # Crear un margen de 20 píxeles alrededor del cuadrado
        x = p[1] * CELL_SIZE + 20
        y = p[0] * CELL_SIZE + 20
        size = CELL_SIZE - 40  # Tamaño reducido del cuadrado
        pygame.draw.rect(WIN, COLOR_POISON, (x, y, size, size))


def main():
    """
    Función principal que ejecuta la demostración interactiva del agente DDQN.

    Esta función implementa un sistema de dos modos que permite a los usuarios
    configurar entornos personalizados y observar el comportamiento del agente.

    Modos de operación:
```

```
    1. **MODO SETUP** (Configuración interactiva):
       - Permite al usuario colocar elementos usando el mouse
       - Clic izquierdo: Añadir frutas (objetivos)
       - Clic derecho: Añadir venenos (obstáculos peligrosos)
       - Barra espaciadora: Iniciar simulación

    2. **MODO RUN** (Demostración del agente):
       - El agente DDQN toma control total
       - Ejecuta acciones basadas en el modelo entrenado
       - Visualización en tiempo real del comportamiento
       - Finalización automática y reinicio

    El sistema está diseñado para demostraciones públicas, permitiendo
    múltiples usuarios configurar y probar diferentes escenarios.
    """
    # Variables de estado del sistema
    fruits = []           # Lista de posiciones de frutas configuradas por el usuario
    poisons = []          # Lista de posiciones de venenos configuradas por el usuario
    mode = "setup"        # Modo inicial: "setup" para configuración, "run" para ejecución

    # Configuración del bucle principal
    clock = pygame.time.Clock()  # Control de framerate
    run = True                   # Flag principal del bucle

    # Bucle principal de la demostración
    while run:
        # Limpiar pantalla con fondo negro
        WIN.fill((0, 0, 0))

        # Dibujar grilla de referencia
        draw_grid()

        # Procesar eventos del usuario
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            # --- LÓGICA DEL MODO SETUP ---
            if mode == "setup":
                # Manejo de clics del mouse para colocar elementos
                if event.type == pygame.MOUSEBUTTONDOWN:
                    pos = pygame.mouse.get_pos()
```

```
                col = pos[0] // CELL_SIZE  # Convertir coordenada x a columna
                row = pos[1] // CELL_SIZE  # Convertir coordenada y a fila

                # Clic izquierdo: Añadir fruta (si no existe ya)
                if event.button == 1 and (row, col) not in fruits:
                    fruits.append((row, col))

                # Clic derecho: Añadir veneno (si no existe ya)
                elif event.button == 3 and (row, col) not in poisons:
                    poisons.append((row, col))

            # Tecla espaciadora: Iniciar simulación
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE:
                    mode = "run"
                    # Configurar el entorno con los elementos del usuario
                    state = env.reset(
                        agent_pos=(0, 0),      # Agente siempre inicia en (0,0)
                        fruit_pos=fruits,      # Frutas configuradas por el usuario
                        poison_pos=poisons     # Venenos configurados por el usuario
                    )
                    print("Iniciando simulación...")

    # --- RENDERIZADO SEGÚN EL MODO ACTUAL ---
    if mode == "setup":
        # Modo configuración: Mostrar elementos colocados por el usuario
        # El agente se dibuja fuera de pantalla (posición inválida) para ocultarlo
        draw_elements(
            np.array([-1, -1]),  # Posición fuera de pantalla para el agente
            fruits,              # Frutas configuradas por el usuario
            poisons              # Venenos configurados por el usuario
        )

    elif mode == "run":
        # --- LÓGICA DEL AGENTE AUTÓNOMO ---
        # Obtener estado actual del entorno
        state = env.get_state()

        # El agente elige la mejor acción sin exploración
        # explore=False asegura que use solo la política aprendida
        action = agent.choose_action(state, explore=False)
```

```
            # Ejecutar la acción en el entorno
            next_state, reward, done = env.step(action)

            # Renderizar estado actual con posiciones reales del entorno
            draw_elements(
                env.agent_pos,    # Posición actual del agente
                env.fruit_pos,    # Frutas restantes en el entorno
                env.poison_pos    # Venenos en el entorno
            )

            # Verificar si el episodio terminó
            if done:
                print("¡Simulación terminada!")
                # Reiniciar sistema para nueva demostración
                fruits = []        # Limpiar frutas configuradas
                poisons = []       # Limpiar venenos configurados
                mode = "setup"     # Volver al modo configuración
                pygame.time.delay(2000)  # Pausa de 2 segundos antes del reinicio

            # Control de velocidad de visualización
            pygame.time.delay(300)  # Pausa de 300ms para observar movimientos

        # Actualizar pantalla para mostrar cambios
        pygame.display.update()

    # Limpiar recursos al salir
    pygame.quit()


if __name__ == "__main__":
    """
    Punto de entrada del programa de demostración.

    Ejecuta la función main() cuando el archivo se ejecuta directamente.
    Este patrón permite importar funciones de este módulo en otros scripts
    sin ejecutar automáticamente la demostración.

    Uso típico:
        python main.py  # Ejecuta la demostración interactiva

    La demostración está diseñada para:
    - Presentaciones públicas del proyecto
```

```
    - Validación rápida del comportamiento del agente
    - Experimentación interactiva con diferentes configuraciones
    - Evaluación cualitativa del rendimiento del modelo
    """
    main()
```

Agente Genético

agente_ga.py

```
# agent_ga.py
"""
Implementación de un agente basado en algoritmos genéticos para el entorno de recolección de

Este módulo define la arquitectura de red neuronal y la clase agente utilizados en el enfoque
de algoritmos genéticos. A diferencia del DQN que aprende mediante gradientes, este agente
evoluciona sus pesos mediante selección natural, mutación y cruzamiento.

Componentes principales:
- AgentNetwork: Red neuronal convolucional para procesar el estado visual
- Agent: Wrapper que contiene la red y maneja la evaluación de fitness

El agente procesa el estado del entorno (representado como una imagen de 3 canales)
y produce directamente acciones sin necesidad de aprendizaje por refuerzo.

"""

import torch
import torch.nn as nn
import numpy as np

# La arquitectura de la red puede ser la misma CNN que ya teníamos.
# Es una buena forma de procesar la "visión" del agente.
class AgentNetwork(nn.Module):
    """
    Red neuronal convolucional para el agente genético.

    Esta red procesa la representación visual del entorno (estado como imagen de 3 canales)
    y produce valores de acción para las 4 direcciones posibles. La arquitectura utiliza
    capas convolucionales para extraer características espaciales, seguidas de capas
    densas para la toma de decisiones.
```

```
    Arquitectura:
    - Conv2D (3→16 canales) + ReLU
    - Conv2D (16→32 canales) + ReLU
    - Flatten
    - Dense (→256) + ReLU
    - Dense (→4 acciones)

    Args:
        h (int): Altura de la cuadrícula de entrada (default: 5)
        w (int): Ancho de la cuadrícula de entrada (default: 5)
        outputs (int): Número de acciones posibles (default: 4)
    """
    def __init__(self, h=5, w=5, outputs=4):
        super(AgentNetwork, self).__init__()

        # Capas convolucionales para procesamiento espacial del estado visual
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)

        def conv2d_size_out(size, kernel_size=3, stride=1, padding=1):
            """
            Calcula el tamaño de salida después de una operación de convolución.
            Formula: (entrada + 2*padding - kernel_size) // stride + 1
            """
            return (size + 2 * padding - kernel_size) // stride + 1

        # Calcular dimensiones para la capa lineal después de las convoluciones
        # Como usamos padding=1 y kernel=3, las dimensiones se mantienen iguales
        convw = conv2d_size_out(conv2d_size_out(w))
        convh = conv2d_size_out(conv2d_size_out(h))
        linear_input_size = convw * convh * 32  # 32 es el número de canales de salida de co

        # Capas densas para la toma de decisiones
        # Capas densas para la toma de decisiones
        self.fc1 = nn.Linear(linear_input_size, 256)  # Capa oculta con 256 neuronas
        self.fc2 = nn.Linear(256, outputs)            # Capa de salida con 4 acciones

    def forward(self, x):
        """
        Propagación hacia adelante de la red neuronal.

        Procesa el estado visual del entorno a través de las capas convolucionales
```

```
        y densas para producir valores de acción.

        Args:
            x (torch.Tensor): Estado del entorno de forma (batch_size, 3, h, w)
                              - Canal 0: Posición del agente
                              - Canal 1: Posiciones de frutas
                              - Canal 2: Posiciones de venenos

        Returns:
            torch.Tensor: Valores de acción de forma (batch_size, 4)
                          Cada valor representa la "utilidad" de una acción:
                          - Índice 0: Arriba
                          - Índice 1: Abajo
                          - Índice 2: Izquierda
                          - Índice 3: Derecha
        """
        # Primera capa convolucional + activación ReLU
        x = nn.functional.relu(self.conv1(x))

        # Segunda capa convolucional + activación ReLU
        x = nn.functional.relu(self.conv2(x))

        # Aplanar tensor para capas densas: (batch, channels*h*w)
        x = x.view(x.size(0), -1)

        # Primera capa densa + activación ReLU
        x = nn.functional.relu(self.fc1(x))

        # Capa de salida (sin activación, valores raw para argmax)
        return self.fc2(x)

# El Agente es ahora solo una cáscara con su red y una puntuación de fitness.
class Agent:
    """
    Wrapper del agente para algoritmos genéticos.

    Esta clase encapsula la red neuronal y proporciona la interfaz necesaria
    para el algoritmo genético. A diferencia de los agentes de RL, este agente
    no aprende durante la ejecución; su comportamiento está completamente
    determinado por los pesos de la red neuronal (sus "genes").

    El agente se evalúa mediante su fitness (rendimiento en el entorno),
```

```
    y los mejores agentes se seleccionan para reproducirse y crear la
    siguiente generación mediante:
    - Selección: Los mejores agentes tienen mayor probabilidad de reproducirse
    - Cruzamiento: Combinación de genes de dos padres
    - Mutación: Cambios aleatorios en los genes

    Attributes:
        network (AgentNetwork): Red neuronal que define el comportamiento del agente
        fitness (float): Puntuación de rendimiento en el entorno (mayor = mejor)
    """
    def __init__(self):
        """
        Inicializa un nuevo agente con red neuronal y fitness en cero.

        Los pesos de la red se inicializan aleatoriamente según la
        inicialización por defecto de PyTorch. Estos pesos representan
        los "genes" del agente que evolucionarán con el tiempo.
        """
        self.network = AgentNetwork()
        self.fitness = 0

    def choose_action(self, state):
        """
        Selecciona una acción basada en el estado actual del entorno.

        El agente utiliza su red neuronal para evaluar el estado y selecciona
        la acción con el valor más alto (estrategia greedy). No hay exploración
        ya que el comportamiento del agente está completamente determinado por
        sus genes (pesos de la red).

        Este método es determinístico: dado el mismo estado y los mismos pesos,
        siempre producirá la misma acción. Esto es importante para la evaluación
        consistente del fitness durante la evolución.

        Args:
            state (np.array): Estado del entorno de forma (3, h, w)
                              - Canal 0: Posición del agente (1.0 donde está, 0.0 resto)
                              - Canal 1: Posiciones de frutas (1.0 donde hay frutas)
                              - Canal 2: Posiciones de venenos (1.0 donde hay venenos)

        Returns:
            int: Acción seleccionada:
```

```
                - 0: Mover arriba (decrementar fila)
                - 1: Mover abajo (incrementar fila)
                - 2: Mover izquierda (decrementar columna)
                - 3: Mover derecha (incrementar columna)
        """
        # Convertir estado NumPy a tensor PyTorch y agregar dimensión de batch
        state_tensor = torch.FloatTensor(state).unsqueeze(0)

        # Evaluación sin gradientes (no hay backpropagation)
        with torch.no_grad():
            action_values = self.network(state_tensor)

        # Seleccionar acción con mayor valor Q (estrategia greedy)
        return torch.argmax(action_values).item()

    def load_genes(self, filepath):
        """
        Carga los "genes" (pesos de la red) desde un archivo.

        Utilizado para cargar agentes previamente evolucionados y demostrar
        su comportamiento. Los pesos representan el "ADN" del agente que
        determina completamente su comportamiento en el entorno.

        Este método es útil para:
        - Cargar el mejor agente de una evolución anterior
        - Demostrar el comportamiento de agentes elite
        - Continuar la evolución desde una generación guardada
        - Análisis y visualización del comportamiento aprendido

        Args:
            filepath (str): Ruta al archivo con los pesos del modelo
                            (normalmente un archivo .pth de PyTorch)

        Raises:
            FileNotFoundError: Si el archivo no existe
            RuntimeError: Si los pesos no coinciden con la arquitectura
        """
        try:
            self.network.load_state_dict(torch.load(filepath))
            print(f"  Genes cargados exitosamente desde: {filepath}")
        except FileNotFoundError:
            print(f"  Error: No se encontró el archivo {filepath}")
```

```
            raise
        except Exception as e:
            print(f"  Error cargando genes: {e}")
            raise
```

genetico_agente.py

```
"""
Demostrador interactivo para agentes entrenados con algoritmos genéticos.

Este módulo implementa una interfaz gráfica completa que permite:
1. Configurar escenarios personalizados con frutas, venenos y paredes
2. Observar el comportamiento de un agente genético entrenado
3. Interactuar en tiempo real con controles de teclado
4. Visualizar el rendimiento del agente en diferentes configuraciones

Características principales:
- Modo Setup: Configuración manual del entorno
- Modo Playing: Demostración del agente en acción
- Controles intuitivos con teclado
- Gráficos mejorados con sprites
- Interfaz informativa con instrucciones

El agente carga pesos previamente evolucionados y demuestra su comportamiento
determinístico en los escenarios configurados por el usuario.

"""

import pygame
import numpy as np
import os
import time
from agent_ga import Agent

# CONFIGURACIÓN VISUAL Y DIMENSIONES
"""
Constantes que definen la apariencia y dimensiones de la interfaz gráfica.
"""
GRID_WIDTH = 5          # Ancho de la cuadrícula en celdas
GRID_HEIGHT = 5         # Alto de la cuadrícula en celdas
CELL_SIZE = 120         # Tamaño de cada celda en píxeles (más grande que en DQN)
```

```
SCREEN_WIDTH = GRID_WIDTH * CELL_SIZE    # Ancho total de la ventana (600px)
SCREEN_HEIGHT = GRID_HEIGHT * CELL_SIZE  # Alto total del área de juego (600px)

# PALETA DE COLORES PROFESIONAL
"""
Esquema de colores dark theme para una interfaz moderna y profesional.
"""
COLOR_FONDO = (25, 25, 25)      # Gris muy oscuro para el fondo
COLOR_LINEAS = (40, 40, 40)     # Gris oscuro para líneas de cuadrícula
COLOR_CURSOR = (255, 255, 0)    # Amarillo brillante para el cursor de selección
COLOR_TEXTO = (230, 230, 230)   # Gris claro para texto legible


class EntornoGrid:
    """
    Entorno de cuadrícula personalizado para la demostración del agente genético.

    Esta clase maneja la lógica del juego y la configuración del entorno,
    incluyendo la colocación de elementos y la simulación de la interacción
    del agente. Incluye características adicionales como paredes que no
    están presentes en los entornos de entrenamiento básicos.

    Características especiales:
    - Soporte para paredes como obstáculos
    - Interfaz de configuración manual
    - Reset automático en condiciones de terminación
    - Estado visual compatible con el agente entrenado

    Attributes:
        size (int): Tamaño de la cuadrícula
        agent_pos (tuple): Posición actual del agente (fila, columna)
        frutas (set): Conjunto de posiciones con frutas
        venenos (set): Conjunto de posiciones con venenos
        paredes (set): Conjunto de posiciones con paredes (obstáculos)
    """
    def __init__(self):
        """
        Inicializa el entorno con configuración vacía.

        Todos los conjuntos de elementos comienzan vacíos, permitiendo
        al usuario configurar el escenario manualmente.
        """
```

```
        self.size = GRID_WIDTH
        self.agent_pos = (0, 0)    # Agente siempre inicia en esquina superior izquierda
        self.frutas = set()        # Conjunto de posiciones de frutas
        self.venenos = set()       # Conjunto de posiciones de venenos
        self.paredes = set()       # Conjunto de posiciones de paredes

    def reset_a_configuracion_inicial(self):
        """
        Resetea el agente a la posición inicial sin modificar el entorno.

        Utilizado al inicio de cada demostración para colocar al agente
        en la posición de partida estándar (0,0) manteniendo la configuración
        de frutas, venenos y paredes establecida por el usuario.

        Returns:
            np.array: Estado inicial del entorno después del reset
        """
        self.agent_pos = (0, 0)
        return self.get_state()

    def limpiar_entorno(self):
        """
        Elimina todos los elementos del entorno (frutas, venenos, paredes).

        Función de utilidad para resetear completamente el escenario,
        permitiendo al usuario comenzar con una cuadrícula vacía.
        El agente permanece en su posición actual.
        """
        self.frutas.clear()
        self.venenos.clear()
        self.paredes.clear()

    def step(self, accion):
        """
        Ejecuta una acción del agente en el entorno de demostración.

        Implementa la lógica del juego incluyendo movimiento, colisiones con
        paredes, interacción con elementos del entorno y cálculo de recompensas.
        Incluye características especiales como paredes que bloquean el movimiento.

        Diferencias con el entorno de entrenamiento:
        - Incluye paredes como obstáculos
```

```
        - Movimientos inválidos dan recompensa negativa
        - Reset automático al completar nivel

        Args:
            accion (int): Acción a ejecutar:
                          0 = Arriba (decrementar fila)
                          1 = Abajo (incrementar fila)
                          2 = Izquierda (decrementar columna)
                          3 = Derecha (incrementar columna)

        Returns:
            tuple: (estado, recompensa, terminado)
                - estado (np.array): Nuevo estado del entorno
                - recompensa (float): Recompensa obtenida
                - terminado (bool): Si el episodio ha terminado
        """
        # Calcular nueva posición basada en la acción
        fila, col = self.agent_pos
        if accion == 0:     # Arriba
            fila -= 1
        elif accion == 1:   # Abajo
            fila += 1
        elif accion == 2:   # Izquierda
            col -= 1
        elif accion == 3:   # Derecha
            col += 1

        # Verificar colisiones: límites del tablero o paredes
        if (
            fila < 0
            or fila >= GRID_HEIGHT
            or col < 0
            or col >= GRID_WIDTH
            or (fila, col) in self.paredes
        ):
            # Movimiento inválido: pequeña penalización, posición no cambia
            return self.get_state(), -0.1, False

        # Movimiento válido: actualizar posición
        self.agent_pos = (fila, col)
        recompensa = -0.05    # Costo base del movimiento
        terminado = False
```

```
        # Verificar interacciones con elementos del entorno
        if self.agent_pos in self.venenos:
            # Veneno tocado: penalización severa y reset a inicio
            recompensa = -10.0
            self.agent_pos = (0, 0)
        elif self.agent_pos in self.frutas:
            # Fruta recogida: recompensa positiva
            recompensa = 1.0
            self.frutas.remove(self.agent_pos)

            # Verificar si se completó el nivel
            if not self.frutas:
                recompensa += 10.0    # Bonus por completar
                terminado = True
                self.agent_pos = (0, 0)  # Reset para próxima demostración

        return self.get_state(), recompensa, terminado

    def get_state(self):
        """
        Genera la representación del estado compatible con el agente entrenado.

        Crea una representación de 3 canales idéntica a la utilizada durante
        el entrenamiento, asegurando compatibilidad con los pesos evolucionados.
        Las paredes no se incluyen en el estado ya que el agente original
        no fue entrenado con ellas.

        Returns:
            np.array: Estado del entorno de forma (3, size, size):
                      - Canal 0: Posición del agente
                      - Canal 1: Posiciones de frutas
                      - Canal 2: Posiciones de venenos
        """
        estado = np.zeros((3, self.size, self.size), dtype=np.float32)

        # Canal 0: Posición del agente
        estado[0, self.agent_pos[0], self.agent_pos[1]] = 1.0

        # Canal 1: Posiciones de frutas
        for fruta in self.frutas:
            estado[1, fruta[0], fruta[1]] = 1.0
```

```
        # Canal 2: Posiciones de venenos
        for veneno in self.venenos:
            estado[2, veneno[0], veneno[1]] = 1.0

        return estado

    def dibujar(
        self,
        pantalla,
        modo_juego,
        cursor_pos,
        img_fruta,
        img_veneno,
        img_pared,
        img_agente,
    ):
        """
        Renderiza el estado completo del entorno en la pantalla.

        Dibuja todos los elementos visuales incluyendo cuadrícula, sprites,
        cursor de selección (en modo setup) e interfaz de usuario con
        controles e información del modo actual.

        Args:
            pantalla (pygame.Surface): Superficie donde dibujar
            modo_juego (str): Modo actual ("SETUP" o "PLAYING")
            cursor_pos (tuple): Posición del cursor en modo setup
            img_fruta (pygame.Surface): Sprite de la fruta
            img_veneno (pygame.Surface): Sprite del veneno
            img_pared (pygame.Surface): Sprite de la pared
            img_agente (pygame.Surface): Sprite del agente
        """
        # Limpiar pantalla con color de fondo
        pantalla.fill(COLOR_FONDO)

        # Dibujar cuadrícula de referencia
        # Líneas verticales
        for x in range(0, SCREEN_WIDTH, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (x, 0), (x, SCREEN_HEIGHT))
        # Líneas horizontales
        for y in range(0, SCREEN_HEIGHT, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (0, y), (SCREEN_WIDTH, y))
```

```
        # Dibujar elementos del entorno (orden importante para superposición correcta)
        # 1. Paredes (fondo)
        for pared in self.paredes:
            pantalla.blit(img_pared, (pared[0] * CELL_SIZE, pared[1] * CELL_SIZE))

        # 2. Frutas
        for fruta in self.frutas:
            pantalla.blit(img_fruta, (fruta[0] * CELL_SIZE, fruta[1] * CELL_SIZE))

        # 3. Venenos
        for veneno in self.venenos:
            pantalla.blit(img_veneno, (veneno[0] * CELL_SIZE, veneno[1] * CELL_SIZE))

        # 4. Agente (primer plano)
        pantalla.blit(
            img_agente, (self.agent_pos[0] * CELL_SIZE, self.agent_pos[1] * CELL_SIZE)
        )

        # 5. Cursor de selección (solo en modo setup)
        if modo_juego == "SETUP":
            cursor_rect = pygame.Rect(
                cursor_pos[0] * CELL_SIZE,
                cursor_pos[1] * CELL_SIZE,
                CELL_SIZE,
                CELL_SIZE,
            )
            pygame.draw.rect(pantalla, COLOR_CURSOR, cursor_rect, 3)

        # Dibujar interfaz de usuario en la parte inferior
        font = pygame.font.Font(None, 24)

        # Información del modo actual
        texto_modo = font.render(f"Modo: {modo_juego}", True, COLOR_TEXTO)

        # Controles disponibles
        controles1 = font.render(
            "SETUP: Flechas, F=Fruta, V=Veneno, W=Pared, C=Limpiar", True, COLOR_TEXTO
        )
        controles2 = font.render("P=Jugar, S=Setup", True, COLOR_TEXTO)

        # Posicionar texto en la parte inferior
        pantalla.blit(texto_modo, (10, SCREEN_HEIGHT + 5))
```

```
        pantalla.blit(controles1, (10, SCREEN_HEIGHT + 30))
        pantalla.blit(controles2, (10, SCREEN_HEIGHT + 55))


def main():
    """
    Función principal que ejecuta la aplicación de demostración.

    Inicializa Pygame, carga recursos gráficos, configura el agente genético
    entrenado y ejecuta el bucle principal de la aplicación. Maneja dos modos
    principales: configuración manual y demostración automática.

    Flujo de ejecución:
    1. Inicialización de Pygame y recursos
    2. Carga del agente entrenado
    3. Bucle principal con manejo de eventos
    4. Renderizado continuo
    5. Limpieza al salir
    """
    # INICIALIZACIÓN DE PYGAME
    pygame.init()
    pantalla = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
    pygame.display.set_caption("Agente Genético - Come Frutas ")

    # FUNCIÓN AUXILIAR PARA CARGA DE IMÁGENES
    def cargar_img(nombre, color_fallback):
        """
        Carga una imagen con fallback a color sólido si falla.

        Args:
            nombre (str): Nombre del archivo de imagen
            color_fallback (tuple): Color RGB de respaldo

        Returns:
            pygame.Surface: Superficie escalada al tamaño de celda
        """
        try:
            ruta = os.path.join(os.path.dirname(__file__), nombre)
            img = pygame.image.load(ruta).convert_alpha()
            return pygame.transform.scale(img, (CELL_SIZE, CELL_SIZE))
        except:
            # Fallback: crear superficie de color sólido
```

```
            surf = pygame.Surface((CELL_SIZE, CELL_SIZE))
            surf.fill(color_fallback)
            return surf

    # CARGA DE RECURSOS GRÁFICOS
    img_fruta = cargar_img("../fruta.png", (0, 255, 0))      # Verde si no hay imagen
    img_veneno = cargar_img("../veneno.png", (255, 0, 0))    # Rojo si no hay imagen
    img_pared = cargar_img("../pared.png", (100, 100, 100)) # Gris si no hay imagen
    img_agente = cargar_img("../agente.png", (0, 0, 255))   # Azul si no hay imagen

    # INICIALIZACIÓN DEL ENTORNO Y AGENTE
    entorno = EntornoGrid()
    agente = Agent()

    # Cargar agente entrenado con algoritmos genéticos
    agente.load_genes("GENETICO/best_agent_genes.pth")

    # VARIABLES DE ESTADO DE LA APLICACIÓN
    cursor_pos = [0, 0]        # Posición del cursor en modo setup
    modo_juego = "SETUP"       # Modo inicial: configuración
    reloj = pygame.time.Clock() # Control de FPS
    corriendo = True           # Flag de control del bucle principal

    # BUCLE PRINCIPAL DE LA APLICACIÓN
    while corriendo:
        # MANEJO DE EVENTOS
        for evento in pygame.event.get():
            if evento.type == pygame.QUIT:
                corriendo = False

            # EVENTOS DE TECLADO
            if evento.type == pygame.KEYDOWN:
                # CONTROLES GLOBALES (disponibles en ambos modos)
                if evento.key == pygame.K_p:
                    print("--- INICIANDO MODO JUEGO ---")
                    entorno.reset_a_configuracion_inicial()
                    modo_juego = "PLAYING"
                    time.sleep(0.5)  # Pausa para transición visual

                elif evento.key == pygame.K_s:
                    print("--- INICIANDO MODO SETUP ---")
                    modo_juego = "SETUP"
```

```
                # CONTROLES ESPECÍFICOS DEL MODO SETUP
                if modo_juego == "SETUP":
                    # Navegación con flechas del cursor
                    if evento.key == pygame.K_UP:
                        cursor_pos[1] = max(0, cursor_pos[1] - 1)
                    elif evento.key == pygame.K_DOWN:
                        cursor_pos[1] = min(GRID_HEIGHT - 1, cursor_pos[1] + 1)
                    elif evento.key == pygame.K_LEFT:
                        cursor_pos[0] = max(0, cursor_pos[0] - 1)
                    elif evento.key == pygame.K_RIGHT:
                        cursor_pos[0] = min(GRID_WIDTH - 1, cursor_pos[0] + 1)

                    # Colocación/eliminación de elementos
                    pos = tuple(cursor_pos)

                    # F = Toggle Fruta
                    if evento.key == pygame.K_f:
                        if pos in entorno.frutas:
                            entorno.frutas.remove(pos)
                            print(f"Fruta eliminada en {pos}")
                        else:
                            entorno.frutas.add(pos)
                            entorno.venenos.discard(pos)    # Remover otros elementos
                            entorno.paredes.discard(pos)
                            print(f"Fruta colocada en {pos}")

                    # V = Toggle Veneno
                    elif evento.key == pygame.K_v:
                        if pos in entorno.venenos:
                            entorno.venenos.remove(pos)
                            print(f"Veneno eliminado en {pos}")
                        else:
                            entorno.venenos.add(pos)
                            entorno.frutas.discard(pos)     # Remover otros elementos
                            entorno.paredes.discard(pos)
                            print(f"Veneno colocado en {pos}")

                    # W = Toggle Pared
                    elif evento.key == pygame.K_w:
                        if pos in entorno.paredes:
                            entorno.paredes.remove(pos)
                            print(f"Pared eliminada en {pos}")
```

```
                    else:
                        entorno.paredes.add(pos)
                        entorno.frutas.discard(pos)     # Remover otros elementos
                        entorno.venenos.discard(pos)
                        print(f"Pared colocada en {pos}")

                # C = Limpiar todo
                elif evento.key == pygame.K_c:
                    print("--- LIMPIANDO ENTORNO COMPLETO ---")
                    entorno.limpiar_entorno()

    # LÓGICA DEL MODO PLAYING (DEMOSTRACIÓN DEL AGENTE)
    if modo_juego == "PLAYING":
        # El agente toma decisiones automáticamente
        estado = entorno.get_state()
        accion = agente.choose_action(estado)
        _, _, terminado = entorno.step(accion)

        # Verificar si el episodio terminó
        if terminado:
            print("  ¡Agente completó el nivel! Volviendo a modo SETUP.")
            modo_juego = "SETUP"

        # Pausa para visualización clara del movimiento
        time.sleep(0.1)

    # RENDERIZADO (COMÚN PARA AMBOS MODOS)
    # Crear superficie temporal para el contenido completo
    pantalla_con_info = pygame.Surface((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
    pantalla_con_info.fill(COLOR_FONDO)

    # Dibujar entorno y elementos
    entorno.dibujar(
        pantalla_con_info,
        modo_juego,
        tuple(cursor_pos),
        img_fruta,
        img_veneno,
        img_pared,
        img_agente,
    )
```

```
        # Copiar a pantalla principal y actualizar
        pantalla.blit(pantalla_con_info, (0, 0))
        pygame.display.flip()

        # Controlar FPS
        reloj.tick(60)

    # LIMPIEZA AL SALIR
    pygame.quit()


if __name__ == "__main__":
    """
    Punto de entrada del programa.

    Ejecuta la función main() solo si este archivo se ejecuta directamente.
    Incluye mensaje de bienvenida con instrucciones básicas.
    """
    print("=" * 60)
    print("  DEMOSTRADOR DE AGENTE GENÉTICO  ")
    print("=" * 60)
    print("CONTROLES:")
    print("  GLOBALES:")
    print("  P - Iniciar modo Playing (demostración)")
    print("  S - Cambiar a modo Setup (configuración)")
    print()
    print("   MODO SETUP:")
    print("      - Mover cursor")
    print("  F - Toggle Fruta")
    print("  V - Toggle Veneno")
    print("  W - Toggle Pared")
    print("  C - Limpiar entorno")
    print()
    print("  MODO PLAYING:")
    print("  El agente toma control automáticamente")
    print("  Observa el comportamiento evolucionado")
    print()
    print("¡Configura un escenario y observa la inteligencia artificial!")
    print("=" * 60)

    main()
```

enviroment.py

```
"""
Entorno de cuadrícula especializado para algoritmos genéticos.

Este módulo implementa un entorno modificado para el entrenamiento de agentes mediante
algoritmos genéticos. La principal diferencia con el entorno DQN es el manejo de venenos:
en lugar de terminar el episodio, el agente es enviado de vuelta a la posición inicial
con una penalización, permitiendo episodios más largos y mejor evaluación de fitness.

Características específicas para GA:
- Venenos no terminan el episodio, sino que resetean la posición
- Episodios más largos para mejor evaluación de fitness
- Recompensas ajustadas para discriminar mejor entre agentes
- Seguimiento de posición inicial para reset de venenos

"""

import numpy as np

class GridEnvironment:
    """
    Entorno de cuadrícula optimizado para algoritmos genéticos.

    Este entorno está diseñado específicamente para la evaluación de agentes
    mediante algoritmos genéticos. La principal modificación es que tocar venenos
    no termina el episodio, sino que envía al agente de vuelta al inicio,
    permitiendo episodios más largos y una mejor discriminación entre agentes.

    Características para GA:
    - Episodios más largos para mejor evaluación de fitness
    - Venenos causan reset de posición en lugar de game over
    - Recompensas ajustadas para mejor selección evolutiva
    - Seguimiento de posición inicial para mecánica de reset

    Attributes:
        size (int): Tamaño de la cuadrícula (size x size)
        start_pos (np.array): Posición inicial del agente en el episodio
        agent_pos (np.array): Posición actual del agente
        fruit_pos (list): Lista de posiciones de frutas
        poison_pos (list): Lista de posiciones de venenos
    """
    def __init__(self, size=5):
```

```
        """
        Inicializa el entorno de cuadrícula para algoritmos genéticos.

        Args:
            size (int, optional): Tamaño de la cuadrícula. Por defecto es 5x5.
        """
        self.size = size
        self.start_pos = (0, 0)  # Guardar posición inicial para reset de venenos
        self.reset()

    def reset(self, agent_pos=(0, 0), fruit_pos=[], poison_pos=[]):
        """
        Reinicia el entorno con una configuración específica.

        Establece las posiciones iniciales y guarda la posición de inicio del agente
        para la mecánica de reset por venenos. Esta posición inicial es crucial
        en el paradigma de algoritmos genéticos ya que permite que el agente
        continúe intentando después de errores.

        Args:
            agent_pos (tuple, optional): Posición inicial del agente (fila, columna).
                                         Por defecto (0, 0).
            fruit_pos (list, optional): Lista de tuplas con posiciones de frutas.
                                        Por defecto lista vacía.
            poison_pos (list, optional): Lista de tuplas con posiciones de venenos.
                                         Por defecto lista vacía.

        Returns:
            np.array: Estado inicial del entorno como array 3D (3, size, size).
        """
        self.start_pos = np.array(agent_pos)  # Guardar posición inicial del episodio
        self.agent_pos = np.array(agent_pos)
        self.fruit_pos = [np.array(p) for p in fruit_pos]
        self.poison_pos = [np.array(p) for p in poison_pos]
        return self.get_state()

    def get_state(self):
        """
        Genera la representación del estado actual del entorno.

        Idéntica implementación al entorno DQN. El estado se representa como una
        "imagen" de 3 canales que puede ser procesada por redes convolucionales.
```

```
        - Canal 0: Posición del agente (1.0 donde está el agente, 0.0 en el resto)
        - Canal 1: Posiciones de frutas (1.0 donde hay frutas, 0.0 en el resto)
        - Canal 2: Posiciones de venenos (1.0 donde hay venenos, 0.0 en el resto)

        Esta representación permite que el agente "vea" todo el entorno de una vez
        y es compatible con arquitecturas de redes neuronales convolucionales.

        Returns:
            np.array: Estado del entorno como array 3D de forma (3, size, size)
                      con valores float32.
        """
        state = np.zeros((3, self.size, self.size), dtype=np.float32)

        # Canal 0: Posición del agente
        # Canal 0: Posición del agente
        state[0, self.agent_pos[0], self.agent_pos[1]] = 1.0

        # Canal 1: Posiciones de las frutas
        for fruit in self.fruit_pos:
            state[1, fruit[0], fruit[1]] = 1.0

        # Canal 2: Posiciones de los venenos
        for poison in self.poison_pos:
            state[2, poison[0], poison[1]] = 1.0

        return state

    def step(self, action):
        """
        Ejecuta una acción en el entorno optimizado para algoritmos genéticos.

        Esta función implementa la lógica principal del juego con modificaciones
        específicas para algoritmos genéticos. La diferencia clave es el manejo
        de venenos: en lugar de terminar el episodio, el agente se resetea a la
        posición inicial, permitiendo episodios más largos y mejor evaluación.

        Diferencias con DQN:
        - Venenos NO terminan el episodio
        - Venenos resetean la posición del agente al inicio
        - Penalización mayor por venenos (-10.0 vs -1.0)
        - Episodios más largos para mejor discriminación de fitness
```

```
        Args:
            action (int): Acción a realizar:
                          0 = Arriba (decrementar fila)
                          1 = Abajo (incrementar fila)
                          2 = Izquierda (decrementar columna)
                          3 = Derecha (incrementar columna)

        Returns:
            tuple: (nuevo_estado, recompensa, terminado)
                - nuevo_estado (np.array): Estado del entorno después de la acción
                - recompensa (float): Recompensa obtenida por la acción
                - terminado (bool): True solo si todas las frutas fueron recogidas
        """

        # FASE 1: MOVIMIENTO DEL AGENTE
        # Lógica idéntica al entorno DQN
        if action == 0:
            self.agent_pos[0] -= 1    # Arriba
        elif action == 1:
            self.agent_pos[0] += 1    # Abajo
        elif action == 2:
            self.agent_pos[1] -= 1    # Izquierda
        elif action == 3:
            self.agent_pos[1] += 1    # Derecha

        # Limitar posición a los límites del tablero
        self.agent_pos = np.clip(self.agent_pos, 0, self.size - 1)

        # FASE 2: INICIALIZACIÓN DE RECOMPENSAS
        reward = -0.05  # Pequeño castigo por cada movimiento
        done = False

        # FASE 2: INICIALIZACIÓN DE RECOMPENSAS
        reward = -0.05  # Pequeño castigo por cada movimiento
        done = False

        # FASE 3: MANEJO ESPECIAL DE VENENOS (DIFERENCIA CLAVE CON DQN)
        if any(np.array_equal(self.agent_pos, p) for p in self.poison_pos):
            # Veneno tocado: penalización severa pero NO termina el episodio
            reward = -10.0
            # CARACTERÍSTICA PRINCIPAL: Reset a posición inicial
            self.agent_pos = np.copy(self.start_pos)
```

```
            # CRÍTICO: done permanece False, el episodio continúa
            print("  Agente tocó veneno, reseteado a posición inicial")
        else:
            # FASE 4: LÓGICA NORMAL (SOLO SI NO HAY VENENO)
            # Verificar si se recogió una fruta
            eaten_fruit_this_step = False
            for i, fruit in enumerate(self.fruit_pos):
                if np.array_equal(self.agent_pos, fruit):
                    reward += 1.0  # Recompensa por fruta
                    self.fruit_pos.pop(i)  # Remover fruta del entorno
                    eaten_fruit_this_step = True
                    print("  Fruta recogida!")
                    break  # Solo una fruta por paso

            # Reward shaping opcional (sin implementar aquí)
            if not eaten_fruit_this_step and self.fruit_pos:
                # Aquí se podría agregar lógica de distancia como en DQN
                # Dejado como comentario para mantener simplicidad
                pass

            # FASE 5: CONDICIÓN DE VICTORIA
            if not self.fruit_pos:
                done = True
                reward += 10.0  # Gran recompensa por completar el nivel
                print("  ¡Todas las frutas recogidas! Episodio completado")

        return self.get_state(), reward, done
```

train_ga.py

```
# train_ga.py
"""
Implementación completa de algoritmo genético para entrenar agentes de IA.

Este módulo implementa un algoritmo genético completo que evoluciona poblaciones
de agentes para resolver el problema de recolección de frutas. El algoritmo
simula la evolución natural mediante selección, cruzamiento y mutación.

Proceso evolutivo:
1. Inicialización: Crear población aleatoria de agentes
2. Evaluación: Probar cada agente en escenarios aleatorios
```

```
3. Selección: Elegir los mejores agentes como padres
4. Cruzamiento: Combinar genes de padres para crear hijos
5. Mutación: Introducir variación aleatoria en los genes
6. Reemplazo: Formar nueva generación con elitismo
7. Repetir hasta convergencia

Características del algoritmo:
- Evaluación en escenarios aleatorios para robustez
- Elitismo para preservar mejores soluciones
- Mutación gaussiana para exploración controlada
- Cruzamiento uniforme para recombinación equilibrada

"""

import torch
import numpy as np
from environment import GridEnvironment
from agent_ga import Agent, AgentNetwork
import random

# HIPERPARÁMETROS DEL ALGORITMO GENÉTICO
"""
Configuración de parámetros evolutivos que controlan el comportamiento
del algoritmo genético. Estos valores han sido ajustados empíricamente
para balancear exploración vs. explotación.
"""
POPULATION_SIZE = 100    # Tamaño de la población por generación
NUM_GENERATIONS = 500    # Número total de generaciones a evolucionar
MUTATION_RATE = 0.05     # Probabilidad de mutación por gen (5%)
ELITISM_COUNT = 25       # Mejores agentes que pasan directamente (25%)

GRID_SIZE = 5           # Tamaño del entorno de evaluación

def create_initial_population():
    """
    Crea la población inicial de agentes con genes aleatorios.

    Genera una población de agentes donde cada uno tiene pesos de red neuronal
    inicializados aleatoriamente según la inicialización por defecto de PyTorch.
    Esta diversidad inicial es crucial para el éxito del algoritmo genético.

    Returns:
```

```
        list: Lista de POPULATION_SIZE agentes con genes aleatorios

    Note:
        La diversidad genética inicial determina el espacio de búsqueda
        que el algoritmo puede explorar durante la evolución.
    """
    return [Agent() for _ in range(POPULATION_SIZE)]

def evaluate_fitness(population, env):
    """
    Evalúa el fitness de cada agente en la población.

    Cada agente se prueba en un escenario aleatorio generado dinámicamente.
    La variabilidad en los escenarios asegura que los agentes desarrollen
    estrategias robustas y generalizables en lugar de sobreajustarse a
    configuraciones específicas.

    Proceso de evaluación:
    1. Generar escenario aleatorio (posiciones, frutas, venenos)
    2. Ejecutar agente por máximo 50 pasos
    3. Acumular recompensas totales como fitness
    4. Repetir para todos los agentes de la población

    Args:
        population (list): Lista de agentes a evaluar
        env (GridEnvironment): Entorno de evaluación

    Note:
        El fitness se calcula como la suma total de recompensas obtenidas
        durante el episodio, incluyendo penalizaciones por movimientos,
        recompensas por frutas y penalizaciones por venenos.
    """
    for agent in population:
        # GENERACIÓN DE ESCENARIO ALEATORIO
        # Número variable de frutas (1-4) para diversidad de dificultad
        num_fruits = np.random.randint(1, 5)

        # Crear lista de todas las posiciones posibles
        all_pos = [(i, j) for i in range(GRID_SIZE) for j in range(GRID_SIZE)]
        random.shuffle(all_pos)  # Barajar para selección aleatoria

        # Asignar posiciones sin solapamiento
```

```
        agent_pos = all_pos.pop()   # Posición inicial del agente
        fruit_pos = [all_pos.pop() for _ in range(num_fruits)]  # Posiciones de frutas
        poison_pos = [all_pos.pop() for _ in range(np.random.randint(1, 4))]  # 1-3 venenos

        # EJECUCIÓN DEL EPISODIO
        state = env.reset(agent_pos=agent_pos, fruit_pos=fruit_pos, poison_pos=poison_pos)
        total_reward = 0

        # Máximo 50 pasos para evitar episodios infinitos
        for _ in range(50):
            action = agent.choose_action(state)
            state, reward, done = env.step(action)
            total_reward += reward

            # Terminar si el agente completa el objetivo
            if done:
                break

        # Asignar fitness como recompensa total acumulada
        agent.fitness = total_reward

def selection(population):
    """
    Selecciona los mejores agentes de la población para reproducción.

    Implementa selección elitista donde solo los agentes con mayor fitness
    se seleccionan como padres para la siguiente generación. Este método
    asegura que las características exitosas se preserven y propaguen.

    Estrategia de selección:
    - Ordenar población por fitness (mayor a menor)
    - Seleccionar el 20% superior como padres
    - Estos padres participarán en cruzamiento y algunos en elitismo

    Args:
        population (list): Población de agentes evaluados

    Returns:
        list: Los mejores agentes seleccionados para reproducción

    Note:
        Un porcentaje del 20% permite suficiente diversidad genética
```

```
        mientras mantiene presión selectiva hacia mejores soluciones.
    """
    # Ordenar por fitness descendente (mejores primero)
    population.sort(key=lambda x: x.fitness, reverse=True)

    # Seleccionar el 20% superior de la población
    return population[:int(POPULATION_SIZE * 0.2)]

def crossover(parent1, parent2):
    """
    Crea un nuevo agente combinando genes de dos padres.

    Implementa cruzamiento uniforme donde cada parámetro (gen) del hijo
    se hereda aleatoriamente de uno de los dos padres. Este método mantiene
    bloques funcionales de la red mientras permite recombinación genética.

    Proceso de cruzamiento:
    1. Crear nuevo agente hijo
    2. Para cada parámetro de la red neuronal:
       - Probabilidad 50%: heredar del padre 1
       - Probabilidad 50%: heredar del padre 2
    3. Cargar genes combinados en el hijo

    Args:
        parent1 (Agent): Primer padre seleccionado
        parent2 (Agent): Segundo padre seleccionado

    Returns:
        Agent: Nuevo agente hijo con genes combinados

    Note:
        El cruzamiento uniforme preserva mejor las estructuras funcionales
        de las redes neuronales comparado con cruzamiento de un punto.
    """
    # Crear nuevo agente hijo
    child = Agent()

    # Obtener diccionarios de parámetros (genes) de los padres
    p1_genes = parent1.network.state_dict()
    p2_genes = parent2.network.state_dict()
    child_genes = child.network.state_dict()
```

```
    # Cruzamiento uniforme: cada gen se hereda aleatoriamente
    for key in child_genes.keys():
        # 50% probabilidad de heredar cada gen de cada padre
        if random.random() < 0.5:
            child_genes[key] = p1_genes[key].clone()  # Heredar del padre 1
        else:
            child_genes[key] = p2_genes[key].clone()  # Heredar del padre 2

    # Cargar genes combinados en la red del hijo
    child.network.load_state_dict(child_genes)
    return child

def mutate(agent):
    """
    Introduce variación genética aleatoria en un agente.

    Implementa mutación gaussiana donde cada parámetro tiene una probabilidad
    MUTATION_RATE de ser alterado con ruido gaussiano. Esta mutación permite
    explorar nuevas regiones del espacio de búsqueda y evitar convergencia
    prematura a óptimos locales.

    Proceso de mutación:
    1. Para cada parámetro de la red neuronal:
       - Probabilidad MUTATION_RATE: agregar ruido gaussiano
       - Magnitud del ruido: distribución normal  =0.1
    2. Recargar parámetros modificados en la red

    Args:
        agent (Agent): Agente a mutar

    Returns:
        Agent: El mismo agente con genes posiblemente mutados

    Note:
        La mutación gaussiana con  =0.1 proporciona un balance entre
        exploración (nuevas soluciones) y explotación (preservar buenas soluciones).
    """
    child_genes = agent.network.state_dict()

    # Aplicar mutación a cada parámetro independientemente
    for key in child_genes.keys():
        if random.random() < MUTATION_RATE:
```

```
            # Agregar ruido gaussiano con desviación estándar 0.1
            noise = torch.randn_like(child_genes[key]) * 0.1
            child_genes[key] += noise

    # Recargar parámetros mutados en la red
    agent.network.load_state_dict(child_genes)
    return agent

if __name__ == "__main__":
    """
    Bucle principal del algoritmo genético.

    Ejecuta el proceso evolutivo completo a través de múltiples generaciones,
    implementando el ciclo: evaluación → selección → cruzamiento → mutación.
    Incluye elitismo para preservar las mejores soluciones y logging detallado
    del progreso evolutivo.
    """
    print("=" * 80)
    print("  INICIANDO ENTRENAMIENTO CON ALGORITMO GENÉTICO  ")
    print("=" * 80)
    print(f"Parámetros de evolución:")
    print(f"  Tamaño de población: {POPULATION_SIZE}")
    print(f"  Generaciones: {NUM_GENERATIONS}")
    print(f"  Tasa de mutación: {MUTATION_RATE*100}%")
    print(f"  Elitismo: {ELITISM_COUNT} agentes")
    print("=" * 80)

    # INICIALIZACIÓN
    env = GridEnvironment()
    population = create_initial_population()

    print("  Población inicial creada")
    print("  Comenzando evolución...")
    print()

    # BUCLE EVOLUTIVO PRINCIPAL
    for gen in range(NUM_GENERATIONS):
        print(f"  Generación {gen+1}/{NUM_GENERATIONS}")

        # FASE 1: EVALUACIÓN DE FITNESS
        evaluate_fitness(population, env)
```

```
        # FASE 2: SELECCIÓN DE PADRES
        parents = selection(population)

        # FASE 3: ANÁLISIS DE PROGRESO
        best_agent_of_gen = parents[0]  # El mejor agente de esta generación
        best_fitness = best_agent_of_gen.fitness
        avg_fitness = np.mean([agent.fitness for agent in population])

        # Logging del progreso evolutivo
        print(f"    Mejor fitness: {best_fitness:.2f}")
        print(f"    Fitness promedio: {avg_fitness:.2f}")
        print(f"    Mejora: {best_fitness - avg_fitness:.2f}")

        # FASE 4: PRESERVACIÓN DEL MEJOR AGENTE
        # Guardar genes del mejor agente de esta generación
        torch.save(best_agent_of_gen.network.state_dict(), "best_agent_genes.pth")
        print(f"    Mejor agente guardado")

        # FASE 5: CREACIÓN DE NUEVA GENERACIÓN
        new_population = []

        # ELITISMO: Los mejores agentes pasan directamente
        new_population.extend(parents[:ELITISM_COUNT])
        print(f"    {ELITISM_COUNT} elite preservados")

        # REPRODUCCIÓN: Llenar resto con descendencia
        offspring_count = 0
        while len(new_population) < POPULATION_SIZE:
            # Seleccionar dos padres aleatoriamente del pool de elite
            parent1, parent2 = random.sample(parents, 2)

            # Cruzamiento: combinar genes de padres
            child = crossover(parent1, parent2)

            # Mutación: introducir variación genética
            child = mutate(child)

            new_population.append(child)
            offspring_count += 1

        print(f"    {offspring_count} descendientes creados")
```

```
        # Reemplazar población anterior
        population = new_population

        print("    Generación completada")
        print("-" * 60)

    # FINALIZACIÓN
    print("\n" + "=" * 80)
    print("  ¡ENTRENAMIENTO EVOLUTIVO COMPLETADO!  ")
    print("=" * 80)
    print("  El mejor ADN evolutivo está guardado en 'best_agent_genes.pth'")
    print("  Puedes usar este agente en los demostradores para ver su comportamiento")
    print("  El agente ha evolucionado a través de", NUM_GENERATIONS, "generaciones")
    print("=" * 80)
```

Imitación

a_star_solver.py

```
# a_star_solver.py
"""
Implementación del algoritmo A* para navegación óptima en grid.

Este módulo proporciona funcionalidades para encontrar el camino más corto
entre dos puntos en una cuadrícula, evitando obstáculos (venenos). Utiliza
el algoritmo A* con distancia Manhattan como heurística para garantizar
optimalidad en entornos de grid.

Funciones:
    heuristic(a, b): Calcula distancia Manhattan entre dos puntos
    a_star_search(grid_size, agent_pos, goal_pos, poisons): Encuentra camino óptimo
"""
import heapq

def heuristic(a, b):
    """
    Calcula la distancia heurística entre dos puntos usando distancia Manhattan.

    La distancia Manhattan es la suma de las diferencias absolutas de sus
    coordenadas cartesianas. Es una heurística admisible para movimiento
    en grid con 4 direcciones, garantizando optimalidad en A*.
```

```
    Args:
        a (tuple): Coordenadas (x, y) del primer punto
        b (tuple): Coordenadas (x, y) del segundo punto

    Returns:
        int: Distancia Manhattan entre los puntos a y b

    Example:
        >>> heuristic((0, 0), (3, 4))
        7
        >>> heuristic((1, 1), (1, 1))
        0
    """
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star_search(grid_size, agent_pos, goal_pos, poisons):
    """
    Implementa el algoritmo A* para encontrar el camino óptimo en una cuadrícula.

    Busca el camino más corto desde la posición del agente hasta el objetivo,
    evitando venenos y respetando los límites del grid. Utiliza una heurística
    admisible (distancia Manhattan) para garantizar optimalidad.

    Algoritmo A*:
        1. Mantiene conjunto abierto (por explorar) y cerrado (explorados)
        2. Evalúa nodos usando f(n) = g(n) + h(n):
           - g(n): Costo real desde inicio hasta nodo n
           - h(n): Heurística desde nodo n hasta objetivo
        3. Expande el nodo con menor f(n) hasta encontrar objetivo
        4. Reconstruye camino desde objetivo hasta inicio

    Args:
        grid_size (int): Tamaño de la cuadrícula (grid_size x grid_size)
        agent_pos (tuple): Posición inicial del agente (x, y)
        goal_pos (tuple): Posición objetivo a alcanzar (x, y)
        poisons (list): Lista de posiciones de venenos [(x, y), ...]

    Returns:
        list: Secuencia de posiciones [(x, y), ...] del camino óptimo,
              excluyendo posición inicial. None si no existe camino.

    Example:
```

```
        >>> a_star_search(5, (0, 0), (4, 4), [(2, 2)])
        [(1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]

    Note:
        - Movimiento limitado a 4 direcciones (arriba, abajo, izquierda, derecha)
        - Venenos son obstáculos impasables
        - Retorna None si el objetivo es inalcanzable
    """
    # Convertir posiciones a tuplas para consistencia
    start = tuple(agent_pos)
    goal = tuple(goal_pos)

    # Inicializar estructuras de datos del algoritmo A*
    close_set = set()           # Nodos ya explorados
    came_from = {}              # Mapeo para reconstruir camino
    g_score = {start: 0}        # Costo real desde inicio
    f_score = {start: heuristic(start, goal)}  # Costo estimado total

    # Heap para mantener nodos ordenados por f_score
    open_heap = [(f_score[start], start)]

    # Bucle principal del algoritmo A*
    while open_heap:
        # Extraer nodo con menor f_score
        current = heapq.heappop(open_heap)[1]

        # ¿Hemos llegado al objetivo?
        if current == goal:
            # Reconstruir camino desde objetivo hasta inicio
            path = [current]
            while current in came_from:
                current = came_from[current]
                path.insert(0, current)
            # Excluir posición inicial del camino retornado
            path.pop(0)
            return path

        # Marcar nodo actual como explorado
        close_set.add(current)

        # Explorar todos los vecinos (4 direcciones)
        for i, j in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
```

```
            neighbor = (current[0] + i, current[1] + j)

            # Verificar límites del grid
            if not (0 <= neighbor[0] < grid_size and 0 <= neighbor[1] < grid_size):
                continue

            # Verificar obstáculos: nodos explorados y venenos
            if neighbor in close_set or any(tuple(p) == neighbor for p in poisons):
                continue

            # Calcular nuevo costo para llegar al vecino
            tentative_g_score = g_score[current] + 1

            # ¿Es este un mejor camino al vecino?
            if tentative_g_score < g_score.get(neighbor, float('inf')):
                # Registrar mejor camino encontrado
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + heuristic(neighbor, goal)
                # Agregar vecino a conjunto de exploración
                heapq.heappush(open_heap, (f_score[neighbor], neighbor))

    # No se encontró camino al objetivo
    return None
```

generate_data.py

```
# generate_data.py
"""
Generador de datos de demostración experta para aprendizaje por imitación.

Este módulo crea datasets de pares estado-acción obtenidos de un agente experto
que utiliza el algoritmo A* para navegación óptima. Los datos generados se
utilizan para entrenar agentes mediante aprendizaje supervisado, imitando
comportamiento experto en diferentes configuraciones de complejidad.

Funciones:
    get_action(from_pos, to_pos): Convierte movimiento posicional a índice de acción
    generate_expert_data_for_n_fruits(num_fruits, num_samples, output_file):
        Genera dataset para configuración específica de frutas
```

```
Constantes:
    GRID_SIZE: Tamaño del entorno de cuadrícula (5x5)
"""
import numpy as np
import random
import pickle
from environment import GridEnvironment
from a_star_solver import a_star_search

# Configuración del entorno
GRID_SIZE = 5

def get_action(from_pos, to_pos):
    """
    Convierte un movimiento entre posiciones adyacentes en índice de acción.

    Calcula la diferencia vectorial entre posiciones y la mapea al índice
    de acción correspondiente. Utilizada para convertir el camino óptimo
    de A* en secuencia de acciones ejecutables por el agente.

    Args:
        from_pos (tuple/np.ndarray): Posición inicial (x, y)
        to_pos (tuple/np.ndarray): Posición objetivo (x, y)

    Returns:
        int: Índice de acción correspondiente al movimiento:
             0 = Arriba (decrementar x)
             1 = Abajo (incrementar x)
             2 = Izquierda (decrementar y)
             3 = Derecha (incrementar y)
             -1 = Movimiento inválido (no adyacente)

    Example:
        >>> get_action((1, 1), (0, 1))  # Movimiento hacia arriba
        0
        >>> get_action((1, 1), (1, 2))  # Movimiento hacia derecha
        3

    Note:
        Solo funciona para posiciones adyacentes. Movimientos diagonales
        o de múltiples celdas retornan -1.
    """
```

```
    # Calcular vector de diferencia entre posiciones
    delta = np.array(to_pos) - np.array(from_pos)

    # Mapear diferencia a índice de acción
    if delta[0] == -1: return 0    # Arriba
    if delta[0] == 1: return 1     # Abajo
    if delta[1] == -1: return 2    # Izquierda
    if delta[1] == 1: return 3     # Derecha
    return -1  # Movimiento inválido

def generate_expert_data_for_n_fruits(num_fruits, num_samples, output_file):
    """
    Genera dataset de demostraciones expertas para configuración específica.

    Crea escenarios aleatorios con número fijo de frutas y utiliza A* para
    generar comportamiento experto óptimo. Implementa estrategia greedy de
    ir siempre a la fruta más cercana, creando datos de entrenamiento para
    aprendizaje por imitación con curriculum learning.

    Proceso de generación:
        1. Crear escenario aleatorio (agente, frutas, venenos)
        2. Calcular fruta más cercana al agente
        3. Usar A* para encontrar camino óptimo
        4. Ejecutar primer paso y registrar par (estado, acción)
        5. Repetir hasta completar episodio o fallar
        6. Continuar hasta obtener muestras suficientes

    Args:
        num_fruits (int): Número de frutas en cada escenario
        num_samples (int): Cantidad objetivo de muestras estado-acción
        output_file (str): Archivo pickle donde guardar el dataset

    Raises:
        IOError: Si no se puede escribir el archivo de salida

    Example:
        >>> generate_expert_data_for_n_fruits(2, 1000, "data_2_fruits.pkl")
        Generando 1000 muestras para 2 fruta(s)...
        Dataset 'data_2_fruits.pkl' creado con 1000 muestras.

    Note:
        - Venenos colocados aleatoriamente (2-4 por escenario)
```

```
        - Máximo 50 pasos por episodio para evitar bucles infinitos
        - Estrategia greedy: siempre ir a fruta más cercana (euclidiana)
        - Solo se registran acciones válidas (con camino A* factible)
    """
    # Inicializar entorno y contenedor de datos
    env = GridEnvironment()
    expert_data = []
    print(f"Generando {num_samples} muestras para {num_fruits} fruta(s)...")

    generated_episodes = 0
    while len(expert_data) < num_samples:
        generated_episodes += 1

        # Configurar escenario aleatorio
        num_poisons = np.random.randint(2, 5)  # 2-4 venenos por escenario

        # Generar posiciones únicas aleatorias
        all_pos = [(r, c) for r in range(GRID_SIZE) for c in range(GRID_SIZE)]
        random.shuffle(all_pos)

        # Asignar posiciones a elementos del entorno
        agent_p = all_pos.pop()
        fruit_p = [all_pos.pop() for _ in range(num_fruits)]
        poison_p = [all_pos.pop() for _ in range(num_poisons)]

        # Inicializar entorno con configuración generada
        env.reset(agent_pos=agent_p, fruit_pos=fruit_p, poison_pos=poison_p)

        # Simular episodio con comportamiento experto
        for _ in range(50):  # Máximo 50 pasos por episodio
            # Verificar condición de terminación por victoria
            if not env.fruit_pos:
                break

            # Implementar estrategia greedy: ir a fruta más cercana
            agent_pos = env.agent_pos
            # Calcular distancias euclidianas a todas las frutas
            distances = [np.linalg.norm(agent_pos - f) for f in env.fruit_pos]
            # Seleccionar fruta más cercana como objetivo
            goal_fruit = env.fruit_pos[np.argmin(distances)]

            # Usar A* para encontrar camino óptimo al objetivo
```

```
            path = a_star_search(GRID_SIZE, agent_pos, goal_fruit, env.poison_pos)

            # Verificar si existe camino factible
            if path and len(path) > 0:
                # Convertir primer paso del camino a acción
                action = get_action(agent_pos, path[0])
                # Registrar par estado-acción para entrenamiento
                state = env.get_state()
                expert_data.append((state, action))
                # Ejecutar acción en el entorno
                env.step(action)
            else:
                # No hay camino factible: terminar episodio
                break

        # Reporte de progreso cada 200 episodios
        if generated_episodes % 200 == 0:
            print(f"  Partidas procesadas: {generated_episodes}, Muestras actuales: {len(exp

    # Guardar dataset en archivo pickle
    with open(output_file, "wb") as f:
        pickle.dump(expert_data[:num_samples], f)
    print(f"Dataset '{output_file}' creado con {len(expert_data[:num_samples])} muestras.")

if __name__ == "__main__":
    """
    Script principal para generación de curriculum de datasets.

    Genera múltiples datasets con diferentes niveles de complejidad para
    implementar curriculum learning en el entrenamiento por imitación.
    Los datasets se ordenan por dificultad creciente (número de frutas).

    Curriculum generado:
        - 1 fruta: 4000 muestras (nivel básico)
        - 2 frutas: 4000 muestras (nivel intermedio bajo)
        - 3 frutas: 4000 muestras (nivel intermedio alto)
        - 4 frutas: 5000 muestras (nivel avanzado, más muestras)

    Beneficios del curriculum learning:
        - Aprendizaje gradual de complejidad creciente
        - Mejor convergencia del entrenamiento
        - Políticas más robustas y generalizables
```

```
        - Reducción de overfitting a configuraciones específicas
    """
    # Generar datasets con complejidad creciente
    generate_expert_data_for_n_fruits(1, 4000, "expert_data_1_fruit.pkl")
    generate_expert_data_for_n_fruits(2, 4000, "expert_data_2_fruits.pkl")
    generate_expert_data_for_n_fruits(3, 4000, "expert_data_3_fruits.pkl")
    generate_expert_data_for_n_fruits(4, 5000, "expert_data_4_fruits.pkl")
    print("\nTodos los datasets del currículo han sido generados.")
```

agent.py

```
# agent.py
"""
Implementación de agente con red neuronal convolucional para aprendizaje por imitación.

Este módulo define la arquitectura de red neuronal y la clase agente utilizadas
en el aprendizaje por imitación. La red procesa representaciones visuales del
entorno (grids 3D) y predice acciones óptimas imitando comportamiento experto.

Clases:
    AgentNetwork: Red neuronal convolucional para procesamiento de estados visuales
    Agent: Interfaz del agente que utiliza la red para toma de decisiones
"""
import torch
import torch.nn as nn

class AgentNetwork(nn.Module):
    """
    Red neuronal convolucional para procesamiento de estados de grid y predicción de acciones

    Arquitectura diseñada específicamente para entornos de cuadrícula donde el estado
    se representa como imágenes de 3 canales (agente, frutas, venenos). Utiliza
    capas convolucionales para extracción de características espaciales seguidas
    de capas densas para predicción de acciones.

    Arquitectura:
        - Conv2D (3->16): Extracción de características básicas
        - Conv2D (16->32): Características de nivel medio
        - Flatten: Preparación para capas densas
        - FC (flattened->256): Representación de alto nivel
        - FC (256->4): Predicción de acciones (4 direcciones)
```

```
    Args:
        h (int): Altura del grid de entrada (default: 5)
        w (int): Ancho del grid de entrada (default: 5)
        outputs (int): Número de acciones posibles (default: 4)
    """
    def __init__(self, h=5, w=5, outputs=4):
        super(AgentNetwork, self).__init__()

        # Capas convolucionales para procesamiento espacial
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)

        # Función auxiliar para calcular tamaño de salida convolucional
        def conv2d_size_out(size, kernel_size=3, stride=1, padding=1):
            """Calcula dimensión de salida después de operación convolucional."""
            return (size + 2 * padding - kernel_size) // stride + 1

        # Calcular dimensiones después de capas convolucionales
        convw = conv2d_size_out(conv2d_size_out(w))
        convh = conv2d_size_out(conv2d_size_out(h))
        linear_input_size = convw * convh * 32

        # Capas densas para predicción final
        self.fc1 = nn.Linear(linear_input_size, 256)
        self.fc2 = nn.Linear(256, outputs)

    def forward(self, x):
        """
        Propagación hacia adelante de la red neuronal.

        Procesa el estado visual del entorno a través de las capas convolucionales
        y densas para generar valores de acción. Utiliza ReLU como función de
        activación para introducir no-linealidad.

        Flujo de procesamiento:
            1. Conv1 + ReLU: Extracción de características básicas
            2. Conv2 + ReLU: Características de nivel medio
            3. Flatten: Conversión a vector 1D
            4. FC1 + ReLU: Representación de alto nivel
            5. FC2: Valores de acción finales (sin activación)

        Args:
```

```
            x (torch.Tensor): Estado del entorno con forma (batch_size, 3, h, w)
                              Canales: [agente, frutas, venenos]

        Returns:
            torch.Tensor: Valores de acción con forma (batch_size, 4)
                          Índices corresponden a [arriba, abajo, izquierda, derecha]
        """
        # Primera capa convolucional con activación ReLU
        x = nn.functional.relu(self.conv1(x))
        # Segunda capa convolucional con activación ReLU
        x = nn.functional.relu(self.conv2(x))
        # Aplanar para conexión con capas densas
        x = x.view(x.size(0), -1)
        # Primera capa densa con activación ReLU
        x = nn.functional.relu(self.fc1(x))
        # Capa de salida sin activación (valores de acción)
        return self.fc2(x)

class Agent:
    """
    Agente que utiliza red neuronal para toma de decisiones por imitación.

    Implementa la interfaz de agente que encapsula la red neuronal y proporciona
    métodos para selección de acciones y carga de modelos pre-entrenados.
    Diseñado para imitar comportamiento experto aprendido de datos de demostración.

    Attributes:
        network (AgentNetwork): Red neuronal convolucional para predicción de acciones
    """
    def __init__(self):
        """
        Inicializa el agente con red neuronal por defecto.

        Crea una instancia de AgentNetwork con parámetros estándar
        para entornos de grid 5x5 con 4 acciones posibles.
        """
        self.network = AgentNetwork()

    def choose_action(self, state):
        """
        Selecciona la acción óptima basada en el estado actual del entorno.
```

```
        Utiliza la red neuronal para evaluar el estado y selecciona la acción
        con mayor valor predicho. Implementa una política determinística
        (greedy) que siempre elige la mejor acción según el modelo.

        Proceso:
            1. Convierte estado a tensor PyTorch
            2. Agrega dimensión de batch (unsqueeze)
            3. Realiza inferencia sin gradientes
            4. Selecciona acción con mayor valor (argmax)

        Args:
            state (numpy.ndarray): Estado del entorno con forma (3, h, w)
                                   Canales: [agente, frutas, venenos]

        Returns:
            int: Índice de la acción seleccionada
                 0=Arriba, 1=Abajo, 2=Izquierda, 3=Derecha

        Note:
            Utiliza torch.no_grad() para optimizar inferencia y evitar
            construcción del grafo computacional durante evaluación.
        """
        # Convertir estado a tensor y agregar dimensión de batch
        state_tensor = torch.FloatTensor(state).unsqueeze(0)

        # Realizar inferencia sin cálculo de gradientes
        with torch.no_grad():
            action_values = self.network(state_tensor)

        # Seleccionar acción con mayor valor predicho
        return torch.argmax(action_values).item()

    def load_model(self, filepath):
        """
        Carga pesos pre-entrenados en la red neuronal del agente.

        Permite cargar modelos entrenados mediante aprendizaje por imitación
        para utilizar políticas aprendidas de datos de demostración experta.
        Los pesos se cargan directamente en la red neuronal existente.

        Args:
            filepath (str): Ruta al archivo de modelo PyTorch (.pth)
```

```
                        que contiene los state_dict de la red

        Raises:
            FileNotFoundError: Si el archivo de modelo no existe
            RuntimeError: Si hay incompatibilidad en arquitectura de red

        Example:
            >>> agent = Agent()
            >>> agent.load_model('imitacion_model.pth')
            >>> action = agent.choose_action(current_state)

        Note:
            El modelo cargado debe tener la misma arquitectura que
            AgentNetwork para evitar errores de compatibilidad.
        """
        self.network.load_state_dict(torch.load(filepath))
```

imitacion_agente.py

```
"""
Demostración interactiva de agente entrenado por aprendizaje por imitación.

Este módulo proporciona una interfaz gráfica para demostrar el comportamiento
de un agente que ha aprendido por imitación de datos expertos. Permite al
usuario diseñar niveles y observar cómo el agente navega utilizando la
política aprendida mediante redes neuronales convolucionales.

Características:
    - Modo configuración: Diseño interactivo de niveles
    - Modo juego: Demostración automática del agente entrenado
    - Interfaz visual: Pygame con sprites y feedback en tiempo real
    - Carga de modelos: Integración con modelos PyTorch pre-entrenados

Constantes:
    GRID_WIDTH, GRID_HEIGHT: Dimensiones del entorno (5x5)
    CELL_SIZE: Tamaño de cada celda en píxeles (120px)
    SCREEN_WIDTH, SCREEN_HEIGHT: Dimensiones de la ventana
    COLOR_*: Esquema de colores para la interfaz

Clases:
    EntornoGrid: Entorno de demostración con funcionalidades completas
```

```
"""
import pygame
import numpy as np
import os
import time
from agent import Agent

# Configuración del entorno y pantalla
GRID_WIDTH = 5
GRID_HEIGHT = 5
CELL_SIZE = 120
SCREEN_WIDTH = GRID_WIDTH * CELL_SIZE
SCREEN_HEIGHT = GRID_HEIGHT * CELL_SIZE

# Esquema de colores para interfaz oscura
COLOR_FONDO = (25, 25, 25)      # Fondo principal oscuro
COLOR_LINEAS = (40, 40, 40)     # Líneas de grid sutiles
COLOR_CURSOR = (255, 255, 0)    # Cursor amarillo brillante
COLOR_TEXTO = (230, 230, 230)   # Texto claro para legibilidad


class EntornoGrid:
    """
    Entorno de demostración para agente entrenado por imitación.

    Implementa un entorno de grid completo con capacidades de configuración
    interactiva y simulación de episodios. Diseñado para demostrar el
    comportamiento aprendido del agente en diferentes escenarios.

    Funcionalidades:
        - Configuración manual de elementos (frutas, venenos, paredes)
        - Simulación de episodios con agente automático
        - Sistema de recompensas completo para feedback
        - Renderizado visual con sprites
        - Detección de colisiones y condiciones de terminación

    Attributes:
        size (int): Tamaño del grid (5x5)
        agent_pos (tuple): Posición actual del agente (fila, columna)
        frutas (set): Conjunto de posiciones de frutas
        venenos (set): Conjunto de posiciones de venenos
        paredes (set): Conjunto de posiciones de paredes
```

```
    """
    def __init__(self):
        """
        Inicializa el entorno con configuración por defecto.

        Establece grid vacío con agente en posición (0,0) y
        conjuntos vacíos para elementos del entorno.
        """
        self.size = GRID_WIDTH
        self.agent_pos = (0, 0)
        self.frutas = set()
        self.venenos = set()
        self.paredes = set()

    def reset_a_configuracion_inicial(self):
        """
        Reinicia el agente a la posición inicial del episodio.

        Coloca al agente en (0,0) manteniendo la configuración actual
        del entorno. Utilizado al iniciar nuevos episodios de demostración.

        Returns:
            np.ndarray: Estado inicial del entorno con forma (3, size, size)
        """
        self.agent_pos = (0, 0)
        return self.get_state()

    def limpiar_entorno(self):
        """
        Elimina todos los elementos del entorno.

        Limpia completamente frutas, venenos y paredes, dejando
        un grid vacío para nueva configuración. El agente mantiene
        su posición actual.
        """
        self.frutas.clear()
        self.venenos.clear()
        self.paredes.clear()

    def step(self, accion):
        """
        Ejecuta una acción del agente y actualiza el estado del entorno.
```

```
        Procesa el movimiento del agente, verifica colisiones y calcula
        recompensas según las interacciones con elementos del entorno.
        Implementa la lógica completa de simulación para demostración.

        Sistema de recompensas:
            - Movimiento normal: -0.05 (costo por paso)
            - Movimiento inválido: -0.1 (penalización)
            - Fruta recolectada: +1.0 (objetivo positivo)
            - Todas las frutas: +10.0 adicional (victoria)
            - Veneno tocado: -10.0 (penalización grave)

        Args:
            accion (int): Acción a ejecutar
                          0 = Arriba (decrementar fila)
                          1 = Abajo (incrementar fila)
                          2 = Izquierda (decrementar columna)
                          3 = Derecha (incrementar columna)

        Returns:
            tuple: (nuevo_estado, recompensa, terminado)
                - nuevo_estado (np.ndarray): Estado resultante (3, size, size)
                - recompensa (float): Recompensa por la acción ejecutada
                - terminado (bool): True si episodio terminó, False en caso contrario
        """
        # Calcular nueva posición basada en la acción
        fila, col = self.agent_pos
        if accion == 0:
            fila -= 1    # Arriba
        elif accion == 1:
            fila += 1    # Abajo
        elif accion == 2:
            col -= 1     # Izquierda
        elif accion == 3:
            col += 1     # Derecha

        # Verificar límites del entorno y colisiones con paredes
        if (
            fila < 0
            or fila >= GRID_HEIGHT
            or col < 0
            or col >= GRID_WIDTH
            or (fila, col) in self.paredes
```

```
        ):
            # Movimiento inválido: mantener posición y penalizar
            return self.get_state(), -0.1, False

        # Movimiento válido: actualizar posición
        self.agent_pos = (fila, col)
        recompensa = -0.05  # Costo base por movimiento
        terminado = False

        # Procesar interacciones con elementos del entorno
        if self.agent_pos in self.venenos:
            # Veneno tocado: penalización grave y reset a inicio
            recompensa = -10.0
            self.agent_pos = (0, 0)
        elif self.agent_pos in self.frutas:
            # Fruta recolectada: recompensa positiva
            recompensa = 1.0
            self.frutas.remove(self.agent_pos)
            # Verificar victoria (todas las frutas recolectadas)
            if not self.frutas:
                recompensa += 10.0  # Bonus por completar nivel
                terminado = True
                self.agent_pos = (0, 0)  # Reset a posición inicial

        return self.get_state(), recompensa, terminado

    def get_state(self):
        """
        Genera representación visual del estado actual del entorno.

        Crea tensor 3D donde cada canal representa un tipo de elemento,
        compatible con la arquitectura CNN del agente entrenado.

        Estructura de canales:
            - Canal 0: Posición del agente (binario)
            - Canal 1: Posiciones de frutas (binario)
            - Canal 2: Posiciones de venenos (binario)

        Returns:
            np.ndarray: Estado con forma (3, size, size) y dtype float32
                        Valores 1.0 indican presencia, 0.0 ausencia
```

```
        Note:
            Las paredes no se incluyen en el estado ya que el agente
            entrenado no las consideraba en los datos de demostración.
        """
        # Inicializar tensor de estado
        estado = np.zeros((3, self.size, self.size), dtype=np.float32)

        # Canal 0: Posición del agente
        estado[0, self.agent_pos[0], self.agent_pos[1]] = 1.0

        # Canal 1: Posiciones de frutas
        for fruta in self.frutas:
            estado[1, fruta[0], fruta[1]] = 1.0

        # Canal 2: Posiciones de venenos
        for veneno in self.venenos:
            estado[2, veneno[0], veneno[1]] = 1.0

        return estado

    def dibujar(
        self,
        pantalla,
        modo_juego,
        cursor_pos,
        img_fruta,
        img_veneno,
        img_pared,
        img_agente,
    ):
        """
        Renderiza el estado completo del entorno con interfaz de usuario.

        Dibuja todos los elementos visuales del entorno, grid de navegación,
        cursor de configuración e información de controles. Proporciona
        feedback visual completo para ambos modos de operación.

        Args:
            pantalla (pygame.Surface): Superficie donde renderizar
            modo_juego (str): Modo actual ("SETUP" o "PLAYING")
            cursor_pos (tuple): Posición del cursor en modo configuración
            img_fruta (pygame.Surface): Sprite de las frutas
```

```
            img_veneno (pygame.Surface): Sprite de los venenos
            img_pared (pygame.Surface): Sprite de las paredes
            img_agente (pygame.Surface): Sprite del agente

        Note:
            Renderiza en orden específico para evitar superposiciones:
            fondo → grid → paredes → frutas → venenos → agente → cursor → UI
        """
        # Limpiar pantalla con fondo oscuro
        pantalla.fill(COLOR_FONDO)

        # Dibujar líneas del grid para navegación visual
        for x in range(0, SCREEN_WIDTH, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (x, 0), (x, SCREEN_HEIGHT))
        for y in range(0, SCREEN_HEIGHT, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (0, y), (SCREEN_WIDTH, y))

        # Renderizar elementos del entorno (orden: paredes → frutas → venenos)
        for pared in self.paredes:
            pantalla.blit(img_pared, (pared[0] * CELL_SIZE, pared[1] * CELL_SIZE))
        for fruta in self.frutas:
            pantalla.blit(img_fruta, (fruta[0] * CELL_SIZE, fruta[1] * CELL_SIZE))
        for veneno in self.venenos:
            pantalla.blit(img_veneno, (veneno[0] * CELL_SIZE, veneno[1] * CELL_SIZE))

        # Dibujar agente (siempre en primer plano)
        pantalla.blit(
            img_agente, (self.agent_pos[0] * CELL_SIZE, self.agent_pos[1] * CELL_SIZE)
        )

        # Mostrar cursor en modo configuración
        if modo_juego == "SETUP":
            cursor_rect = pygame.Rect(
                cursor_pos[0] * CELL_SIZE,
                cursor_pos[1] * CELL_SIZE,
                CELL_SIZE,
                CELL_SIZE,
            )
            pygame.draw.rect(pantalla, COLOR_CURSOR, cursor_rect, 3)

        # Renderizar información de interfaz
        font = pygame.font.Font(None, 24)
```

```
        texto_modo = font.render(f"Modo: {modo_juego}", True, COLOR_TEXTO)
        controles1 = font.render(
            "SETUP: Flechas, F=Fruta, V=Veneno, W=Pared, C=Limpiar", True, COLOR_TEXTO
        )
        controles2 = font.render("P=Jugar, S=Setup", True, COLOR_TEXTO)
        pantalla.blit(texto_modo, (10, SCREEN_HEIGHT + 5))
        pantalla.blit(controles1, (10, SCREEN_HEIGHT + 30))
        pantalla.blit(controles2, (10, SCREEN_HEIGHT + 55))


def main():
    """
    Función principal de la demostración interactiva del agente por imitación.

    Inicializa la interfaz gráfica y gestiona el bucle principal que permite
    alternar entre modo configuración (diseño de niveles) y modo demostración
    (agente automático). Proporciona una experiencia completa para evaluar
    el rendimiento del agente entrenado.

    Flujo de la aplicación:
        1. Inicialización de Pygame y recursos
        2. Carga del modelo entrenado
        3. Bucle principal con dos modos:
           - SETUP: Configuración manual de niveles
           - PLAYING: Demostración automática del agente
        4. Manejo de eventos y renderizado en tiempo real

    Controles disponibles:
        Modo SETUP:
            - Flechas: Mover cursor de configuración
            - F: Colocar/quitar fruta
            - V: Colocar/quitar veneno
            - W: Colocar/quitar pared
            - C: Limpiar entorno completamente
            - P: Iniciar demostración automática

        Modo PLAYING:
            - S: Volver a modo configuración
            - Agente se mueve automáticamente cada 0.1 segundos

    Note:
        Requiere modelo entrenado en "IMITACION/imitacion_model.pth"
```

```
    y sprites en directorio padre (../fruta.png, etc.)
"""
# Inicializar Pygame y configurar ventana
pygame.init()
pantalla = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
pygame.display.set_caption("Agente por Imitación - Come Frutas  ")

def cargar_img(nombre, color_fallback):
    """
    Función auxiliar para carga robusta de sprites.

    Intenta cargar imagen desde archivo, si falla crea superficie
    de color sólido como respaldo para mantener funcionalidad.

    Args:
        nombre (str): Nombre del archivo de imagen
        color_fallback (tuple): Color RGB de respaldo

    Returns:
        pygame.Surface: Sprite cargado o superficie de color
    """
    try:
        ruta = os.path.join(os.path.dirname(__file__), nombre)
        img = pygame.image.load(ruta).convert_alpha()
        return pygame.transform.scale(img, (CELL_SIZE, CELL_SIZE))
    except:
        surf = pygame.Surface((CELL_SIZE, CELL_SIZE))
        surf.fill(color_fallback)
        return surf

# Cargar sprites con colores de respaldo
img_fruta = cargar_img("../fruta.png", (0, 255, 0))        # Verde
img_veneno = cargar_img("../veneno.png", (255, 0, 0))      # Rojo
img_pared = cargar_img("../pared.png", (100, 100, 100))    # Gris
img_agente = cargar_img("../agente.png", (0, 0, 255))      # Azul

# Inicializar entorno y agente
entorno = EntornoGrid()
agente = Agent()
agente.load_model("IMITACION/imitacion_model.pth")

# Variables de estado de la aplicación
```

```
    cursor_pos = [0, 0]
    modo_juego = "SETUP"
    reloj = pygame.time.Clock()
    corriendo = True

    # Bucle principal de la aplicación
    while corriendo:
        # Procesar eventos de entrada
        for evento in pygame.event.get():
            if evento.type == pygame.QUIT:
                corriendo = False

            if evento.type == pygame.KEYDOWN:
                # Cambio de modos globales
                if evento.key == pygame.K_p:
                    print("--- MODO JUEGO ---")
                    entorno.reset_a_configuracion_inicial()
                    modo_juego = "PLAYING"
                    time.sleep(0.5)  # Pausa para visibilidad del cambio

                elif evento.key == pygame.K_s:
                    print("--- MODO SETUP ---")
                    modo_juego = "SETUP"

                # Controles específicos del modo SETUP
                if modo_juego == "SETUP":
                    # Navegación del cursor con flechas
                    if evento.key == pygame.K_UP:
                        cursor_pos[1] = max(0, cursor_pos[1] - 1)
                    elif evento.key == pygame.K_DOWN:
                        cursor_pos[1] = min(GRID_HEIGHT - 1, cursor_pos[1] + 1)
                    elif evento.key == pygame.K_LEFT:
                        cursor_pos[0] = max(0, cursor_pos[0] - 1)
                    elif evento.key == pygame.K_RIGHT:
                        cursor_pos[0] = min(GRID_WIDTH - 1, cursor_pos[0] + 1)

                    # Colocación/eliminación de elementos
                    pos = tuple(cursor_pos)
                    if evento.key == pygame.K_f:
                        # Toggle fruta: agregar/quitar y limpiar otros elementos
                        if pos in entorno.frutas:
                            entorno.frutas.remove(pos)
```

```
                    else:
                        entorno.frutas.add(pos)
                        entorno.venenos.discard(pos)
                        entorno.paredes.discard(pos)
                elif evento.key == pygame.K_v:
                    # Toggle veneno: agregar/quitar y limpiar otros elementos
                    if pos in entorno.venenos:
                        entorno.venenos.remove(pos)
                    else:
                        entorno.venenos.add(pos)
                        entorno.frutas.discard(pos)
                        entorno.paredes.discard(pos)
                elif evento.key == pygame.K_w:
                    # Toggle pared: agregar/quitar y limpiar otros elementos
                    if pos in entorno.paredes:
                        entorno.paredes.remove(pos)
                    else:
                        entorno.paredes.add(pos)
                        entorno.frutas.discard(pos)
                        entorno.venenos.discard(pos)
                elif evento.key == pygame.K_c:
                    # Limpiar entorno completamente
                    print("--- LIMPIANDO ENTORNO ---")
                    entorno.limpiar_entorno()

        # Lógica del modo PLAYING (agente automático)
        if modo_juego == "PLAYING":
            # Obtener estado actual y decidir acción
            estado = entorno.get_state()
            accion = agente.choose_action(estado)
            # Ejecutar acción y verificar terminación
            _, _, terminado = entorno.step(accion)
            if terminado:
                print("Juego terminado. Volviendo a SETUP.")
                modo_juego = "SETUP"
            time.sleep(0.1)  # Velocidad de demostración controlada

        # Renderizado del estado actual
        pantalla_con_info = pygame.Surface((SCREEN_WIDTH, SCREEN_HEIGHT + 80))
        pantalla_con_info.fill(COLOR_FONDO)
        entorno.dibujar(
            pantalla_con_info,
```

```
            modo_juego,
            tuple(cursor_pos),
            img_fruta,
            img_veneno,
            img_pared,
            img_agente,
        )
        pantalla.blit(pantalla_con_info, (0, 0))
        pygame.display.flip()
        reloj.tick(60)  # 60 FPS para fluidez visual

    # Limpiar recursos al salir
    pygame.quit()


if __name__ == "__main__":
    main()
```

train_curriculum.py

```
# train_curriculum.py
"""
Entrenamiento por curriculum learning para aprendizaje por imitación.

Este módulo implementa el entrenamiento de la red neuronal convolucional
utilizando curriculum learning con datasets de complejidad creciente.
El entrenamiento progresa desde escenarios simples (1 fruta) hasta
complejos (4 frutas), mejorando la convergencia y generalización.

Características:
    - Curriculum learning con 4 niveles de dificultad
    - Entrenamiento supervisado con pares estado-acción
    - Optimización Adam con learning rate adaptado
    - CrossEntropyLoss para clasificación de acciones
    - Progresión gradual de épocas por complejidad

Constantes:
    LEARNING_RATE: Tasa de aprendizaje para optimizador Adam (0.0005)
    BATCH_SIZE: Tamaño de lote para entrenamiento (128 muestras)
    CURRICULUM: Secuencia de datasets y épocas de entrenamiento
```

```
Flujo del entrenamiento:
    1. Lección 1: 1 fruta → 25 épocas (fundamentos básicos)
    2. Lección 2: 2 frutas → 30 épocas (navegación intermedia)
    3. Lección 3: 3 frutas → 40 épocas (planificación compleja)
    4. Lección 4: 4 frutas → 50 épocas (maestría y refinamiento)
"""
import torch
import torch.nn as nn
import torch.optim as optim
import pickle
import numpy as np
from agent import AgentNetwork

# Hiperparámetros de entrenamiento
LEARNING_RATE = 0.0005  # Tasa de aprendizaje conservadora para estabilidad
BATCH_SIZE = 128        # Tamaño de lote balanceado para memoria y convergencia

# Curriculum de entrenamiento: (archivo_dataset, num_épocas)
CURRICULUM = [
    ("expert_data_1_fruit.pkl", 25),   # Nivel básico: conceptos fundamentales
    ("expert_data_2_fruits.pkl", 30),  # Nivel intermedio: decisiones múltiples
    ("expert_data_3_fruits.pkl", 40),  # Nivel avanzado: planificación compleja
    ("expert_data_4_fruits.pkl", 50)   # Nivel experto: refinamiento y maestría
]

if __name__ == "__main__":
    """
    Script principal de entrenamiento por curriculum learning.

    Implementa el entrenamiento secuencial de la red neuronal utilizando
    datasets de complejidad creciente. Cada lección del curriculum se
    enfoca en un nivel específico de dificultad, permitiendo al modelo
    aprender gradualmente conceptos más complejos.

    Proceso de entrenamiento:
        1. Inicialización del modelo, optimizador y función de pérdida
        2. Para cada lección del curriculum:
           a. Cargar dataset correspondiente
           b. Preparar DataLoader con batches mezclados
           c. Entrenar por número específico de épocas
           d. Monitorear pérdida promedio por época
        3. Guardar modelo final entrenado
```

```
    Beneficios del curriculum learning:
        - Convergencia más rápida y estable
        - Mejor generalización a nuevos escenarios
        - Reducción de overfitting a configuraciones específicas
        - Aprendizaje progresivo de conceptos complejos
    """
    # Inicializar componentes del entrenamiento
    model = AgentNetwork()                                  # Red convolucional para predicción d
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)  # Optimizador Adam para gra
    criterion = nn.CrossEntropyLoss()                      # Función de pérdida para clasificaci

    # Ejecutar curriculum learning secuencial
    for i, (dataset_file, num_epochs) in enumerate(CURRICULUM):
        print(f"\n--- Iniciando Lección {i+1}/{len(CURRICULUM)}: {dataset_file} ---")

        # Cargar dataset de demostración experta
        with open(dataset_file, "rb") as f:
            data = pickle.load(f)

        # Preparar datos para entrenamiento
        # Separar estados (entrada CNN) y acciones (etiquetas de clasificación)
        states = torch.FloatTensor(np.array([item[0] for item in data]))   # Estados visuales
        actions = torch.LongTensor(np.array([item[1] for item in data]))   # Índices de acci

        # Crear DataLoader para entrenamiento por lotes
        dataloader = torch.utils.data.DataLoader(
            torch.utils.data.TensorDataset(states, actions),
            batch_size=BATCH_SIZE,
            shuffle=True  # Mezclar datos para evitar patrones de orden
        )

        # Entrenar modelo en el dataset actual
        for epoch in range(num_epochs):
            epoch_loss = 0.0  # Acumulador de pérdida para la época

            # Procesar todos los lotes del dataset
            for batch_states, batch_actions in dataloader:
                # Paso hacia adelante: predicción del modelo
                optimizer.zero_grad()           # Limpiar gradientes acumulados
                outputs = model(batch_states)   # Inferencia: estados → valores de acción

                # Calcular pérdida de clasificación
```

```
                loss = criterion(outputs, batch_actions)  # CrossEntropy entre predicción y e

                # Retropropagación y optimización
                loss.backward()    # Calcular gradientes por backpropagation
                optimizer.step()   # Actualizar pesos de la red

                # Acumular pérdida para monitoreo
                epoch_loss += loss.item()

            # Reporte de progreso por época
            avg_loss = epoch_loss / len(dataloader)
            print(f"  Época {epoch+1}/{num_epochs}, Pérdida: {avg_loss:.4f}")

    # Guardar modelo entrenado final
    torch.save(model.state_dict(), "imitacion_model.pth")
    print("\n¡Entrenamiento por currículo completado! Modelo final guardado.")
```

environment.py

```
# environment.py
"""
Entorno de cuadrícula para aprendizaje por imitación de agentes.

Este módulo implementa un entorno de grid simplificado donde un agente
debe navegar para recolectar frutas mientras evita venenos. Está diseñado
específicamente para generar datos de demostración experta y entrenar
agentes mediante aprendizaje por imitación.

Clases:
    GridEnvironment: Entorno de cuadrícula con estados visuales 3D
"""
import numpy as np

class GridEnvironment:
    """
    Entorno de cuadrícula para simulación de navegación y recolección.

    Implementa un mundo de grid 2D donde el agente debe recolectar todas
    las frutas evitando venenos. El estado se representa como una imagen
    de 3 canales (agente, frutas, venenos) ideal para redes convolucionales.
```

```
    Características:
        - Grid cuadrado de tamaño configurable
        - Estados visuales como tensores 3D
        - Movimiento con límites del entorno
        - Detección automática de colisiones
        - Condiciones de terminación por victoria/derrota

    Attributes:
        size (int): Tamaño del grid (size x size)
        agent_pos (np.ndarray): Posición actual del agente [x, y]
        fruit_pos (list): Lista de posiciones de frutas [np.ndarray, ...]
        poison_pos (list): Lista de posiciones de venenos [np.ndarray, ...]
    """
    def __init__(self, size=5):
        """
        Inicializa el entorno de grid con tamaño especificado.

        Args:
            size (int): Dimensión del grid cuadrado (default: 5)
        """
        self.size = size
        self.reset()

    def reset(self, agent_pos=(0, 0), fruit_pos=[], poison_pos=[]):
        """
        Reinicia el entorno con configuración específica de elementos.

        Establece posiciones iniciales del agente, frutas y venenos.
        Utilizado para crear escenarios específicos para generación
        de datos de demostración o evaluación de políticas.

        Args:
            agent_pos (tuple): Posición inicial del agente (x, y) (default: (0,0))
            fruit_pos (list): Lista de posiciones de frutas [(x,y), ...] (default: [])
            poison_pos (list): Lista de posiciones de venenos [(x,y), ...] (default: [])

        Returns:
            np.ndarray: Estado inicial del entorno con forma (3, size, size)

        Note:
            Las listas de posiciones se convierten a arrays numpy para
            operaciones vectorizadas eficientes durante la simulación.
```

```
        """
        self.agent_pos = np.array(agent_pos)
        self.fruit_pos = [np.array(p) for p in fruit_pos]
        self.poison_pos = [np.array(p) for p in poison_pos]
        return self.get_state()

    def get_state(self):
        """
        Genera representación visual del estado actual como tensor 3D.

        Crea una imagen de 3 canales donde cada canal representa un tipo
        de elemento del entorno. Esta representación es ideal para redes
        convolucionales que procesan información espacial.

        Estructura de canales:
            - Canal 0: Posición del agente (binario)
            - Canal 1: Posiciones de frutas (binario)
            - Canal 2: Posiciones de venenos (binario)

        Returns:
            np.ndarray: Estado con forma (3, size, size) y dtype float32
                        Valores: 1.0 para presencia de elemento, 0.0 para ausencia

        Example:
            Para grid 3x3 con agente en (0,0) y fruta en (1,1):
            Canal 0: [[1, 0, 0],    Canal 1: [[0, 0, 0],    Canal 2: [[0, 0, 0],
                      [0, 0, 0],              [0, 1, 0],              [0, 0, 0],
                      [0, 0, 0]]              [0, 0, 0]]              [0, 0, 0]]
        """
        # Inicializar tensor de estado con ceros
        state = np.zeros((3, self.size, self.size), dtype=np.float32)

        # Canal 0: Posición del agente
        state[0, self.agent_pos[0], self.agent_pos[1]] = 1.0

        # Canal 1: Posiciones de frutas
        for fruit in self.fruit_pos:
            state[1, fruit[0], fruit[1]] = 1.0

        # Canal 2: Posiciones de venenos
        for poison in self.poison_pos:
            state[2, poison[0], poison[1]] = 1.0
```

```
        return state

    def step(self, action):
        """
        Ejecuta una acción en el entorno y actualiza el estado.

        Procesa el movimiento del agente, maneja colisiones con límites,
        detecta recolección de frutas y verifica condiciones de terminación.
        Implementa la lógica core del entorno para simulación de episodios.

        Flujo de ejecución:
            1. Actualizar posición según acción
            2. Aplicar límites del entorno
            3. Procesar recolección de frutas
            4. Verificar colisiones con venenos
            5. Evaluar condiciones de terminación

        Args:
            action (int): Acción a ejecutar
                          0 = Arriba (decrementar x)
                          1 = Abajo (incrementar x)
                          2 = Izquierda (decrementar y)
                          3 = Derecha (incrementar y)

        Returns:
            tuple: (nuevo_estado, reward, done)
                - nuevo_estado (np.ndarray): Estado resultante (3, size, size)
                - reward (float): Recompensa por la acción (-0.1 por defecto)
                - done (bool): True si episodio terminó, False en caso contrario

        Note:
            El reward no se utiliza en aprendizaje por imitación pero se
            mantiene para compatibilidad con interfaces de RL estándar.
        """
        # Actualizar posición del agente según la acción
        if action == 0:
            self.agent_pos[0] -= 1  # Arriba
        elif action == 1:
            self.agent_pos[0] += 1  # Abajo
        elif action == 2:
            self.agent_pos[1] -= 1  # Izquierda
        elif action == 3:
```

```
            self.agent_pos[1] += 1  # Derecha

        # Aplicar límites del entorno (clipping)
        self.agent_pos = np.clip(self.agent_pos, 0, self.size - 1)

        # Inicializar variables de terminación
        done = False
        reward = -0.1  # Penalización por paso (no usado en imitación)

        # Verificar recolección de frutas
        for i, fruit in enumerate(self.fruit_pos):
            if np.array_equal(self.agent_pos, fruit):
                # Fruta recolectada: eliminar de la lista
                self.fruit_pos.pop(i)
                break

        # Verificar colisión con venenos (derrota)
        if any(np.array_equal(self.agent_pos, p) for p in self.poison_pos):
            done = True

        # Verificar victoria (todas las frutas recolectadas)
        if not self.fruit_pos:
            done = True

        return self.get_state(), reward, done
```

main.py

```
# main.py
"""
Demostración simple del agente entrenado por aprendizaje por imitación.

Este módulo proporciona una interfaz minimalista para configurar escenarios
y observar el comportamiento del agente entrenado. Utiliza formas geométricas
simples para representar elementos, enfocándose en la funcionalidad core
sin distracciones visuales complejas.

Características:
    - Configuración interactiva con mouse (clic izquierdo=fruta, clic derecho=veneno)
    - Demostración automática del agente entrenado
    - Representación visual simple con formas geométricas
```

```
    - Ciclo continuo configuración → demostración → reset

Constantes:
    GRID_SIZE: Tamaño del entorno (5x5)
    CELL_SIZE: Tamaño de cada celda en píxeles (100px)
    WIDTH, HEIGHT: Dimensiones de la ventana (500x500)
    COLOR_*: Esquema de colores para elementos visuales

Funciones:
    draw_elements: Renderizado de elementos con formas geométricas
    main: Bucle principal con modos configuración y demostración
"""
import pygame
import numpy as np
from environment import GridEnvironment
from agent import Agent

# Configuración del entorno y ventana
GRID_SIZE = 5
CELL_SIZE = 100
WIDTH, HEIGHT = GRID_SIZE * CELL_SIZE, GRID_SIZE * CELL_SIZE

# Esquema de colores simple y claro
COLOR_GRID = (200, 200, 200)    # Gris claro para grid
COLOR_AGENT = (0, 0, 255)       # Azul para agente
COLOR_FRUIT = (0, 255, 0)       # Verde para frutas
COLOR_POISON = (255, 0, 0)      # Rojo para venenos

def draw_elements(win, agent_pos, fruits, poisons):
    """
    Renderiza todos los elementos del entorno usando formas geométricas simples.

    Dibuja el grid de navegación y representa cada elemento del entorno
    con formas distintivas: rectángulos para agente, círculos para frutas
    y cuadrados pequeños para venenos. Diseño minimalista para claridad.

    Representación visual:
        - Agente: Rectángulo azul de celda completa
        - Frutas: Círculos verdes centrados (1/3 del tamaño de celda)
        - Venenos: Cuadrados rojos con margen (80% del tamaño de celda)
        - Grid: Líneas grises para delimitación de celdas
```

```
    Args:
        win (pygame.Surface): Superficie donde renderizar
        agent_pos (np.ndarray): Posición del agente [fila, columna]
        fruits (list): Lista de posiciones de frutas [(fila, col), ...]
        poisons (list): Lista de posiciones de venenos [(fila, col), ...]

    Note:
        Convierte coordenadas (fila, columna) a píxeles (x, y) para Pygame.
        Agente en posición (-1, -1) no se dibuja (modo configuración).
    """
    # Limpiar pantalla con fondo negro
    win.fill((0,0,0))

    # Dibujar grid de navegación
    for x in range(0, WIDTH, CELL_SIZE):
        pygame.draw.line(win, COLOR_GRID, (x, 0), (x, HEIGHT))
    for y in range(0, HEIGHT, CELL_SIZE):
        pygame.draw.line(win, COLOR_GRID, (0, y), (WIDTH, y))
    # Dibujar agente (solo si posición válida)
    if agent_pos[0] >= 0 and agent_pos[1] >= 0:
        pygame.draw.rect(win, COLOR_AGENT,
                         (agent_pos[1] * CELL_SIZE, agent_pos[0] * CELL_SIZE,
                          CELL_SIZE, CELL_SIZE))

    # Dibujar frutas como círculos verdes
    for f in fruits:
        center_x = f[1] * CELL_SIZE + CELL_SIZE//2
        center_y = f[0] * CELL_SIZE + CELL_SIZE//2
        radius = CELL_SIZE//3
        pygame.draw.circle(win, COLOR_FRUIT, (center_x, center_y), radius)

    # Dibujar venenos como cuadrados rojos con margen
    for p in poisons:
        margin = 20  # Margen de 20px para distinguir de agente
        rect_x = p[1] * CELL_SIZE + margin
        rect_y = p[0] * CELL_SIZE + margin
        rect_size = CELL_SIZE - 2 * margin
        pygame.draw.rect(win, COLOR_POISON, (rect_x, rect_y, rect_size, rect_size))

    # Actualizar display para mostrar cambios
    pygame.display.update()
```

```
def main():
    """
    Función principal de la demostración simple del agente por imitación.

    Implementa un ciclo de dos modos: configuración interactiva donde el usuario
    coloca elementos con el mouse, y demostración automática donde el agente
    entrenado navega el escenario. Diseñado para evaluación rápida y directa
    del rendimiento del modelo.

    Flujo de la aplicación:
        1. Modo "setup": Usuario configura escenario con mouse
           - Clic izquierdo: Colocar fruta
           - Clic derecho: Colocar veneno
           - Espacio: Iniciar demostración

        2. Modo "run": Agente navega automáticamente
           - Inferencia con modelo entrenado
           - Movimiento automático cada 300ms
           - Terminación por victoria/derrota
           - Reset automático a configuración

    Controles:
        - Clic izquierdo: Agregar fruta en posición del mouse
        - Clic derecho: Agregar veneno en posición del mouse
        - Espacio: Iniciar demostración (solo si hay frutas)
        - Automático: Reset a configuración al terminar episodio

    Note:
        Requiere modelo entrenado "imitacion_model.pth" en directorio actual.
        El agente siempre inicia en posición (0,0) del grid.
    """
    # Inicializar Pygame y configurar ventana
    pygame.init()
    win = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Agente Come-Frutas (IA)")

    # Inicializar entorno y agente con modelo pre-entrenado
    env = GridEnvironment(size=GRID_SIZE)
    agent = Agent()
    agent.load_model("imitacion_model.pth")

    # Variables de estado de la aplicación
```

```
    fruits, poisons = [], []  # Listas de posiciones de elementos
    mode = "setup"           # Modo inicial: configuración
    run = True               # Control del bucle principal
    # Bucle principal de la aplicación
    while run:
        # Procesar eventos de entrada
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            # Lógica específica del modo configuración
            if mode == "setup":
                if event.type == pygame.MOUSEBUTTONDOWN:
                    # Convertir posición del mouse a coordenadas de grid
                    pos = pygame.mouse.get_pos()
                    col, row = pos[0] // CELL_SIZE, pos[1] // CELL_SIZE

                    # Clic izquierdo: Agregar fruta (si no existe)
                    if event.button == 1 and (row, col) not in fruits:
                        fruits.append((row, col))
                    # Clic derecho: Agregar veneno (si no existe)
                    elif event.button == 3 and (row, col) not in poisons:
                        poisons.append((row, col))

                # Espacio: Iniciar demostración si hay frutas configuradas
                if event.type == pygame.KEYDOWN and event.key == pygame.K_SPACE:
                    if fruits:  # Solo si hay al menos una fruta
                        mode = "run"
                        # Inicializar entorno con configuración actual
                        env.reset(agent_pos=(0,0), fruit_pos=fruits, poison_pos=poisons)

        # Renderizado según el modo actual
        if mode == "setup":
            # Modo configuración: mostrar elementos sin agente
            draw_elements(win, np.array([-1,-1]), fruits, poisons)

        elif mode == "run":
            # Modo demostración: agente automático

            # Obtener estado actual y generar acción
            state = env.get_state()
            action = agent.choose_action(state)
```

```
            # Ejecutar acción y verificar terminación
            _, _, done = env.step(action)

            # Renderizar estado actualizado
            draw_elements(win, env.agent_pos, env.fruit_pos, env.poison_pos)

            # Procesar terminación del episodio
            if done:
                print("¡Simulación terminada!")
                pygame.time.delay(2000)  # Pausa para observar resultado final
                # Reset automático a modo configuración
                fruits, poisons = [], []
                mode = "setup"

            # Controlar velocidad de demostración
            pygame.time.delay(300)  # 300ms entre acciones para visibilidad

    # Limpiar recursos al salir
    pygame.quit()


if __name__ == "__main__":
    main()
```

Jugador humano

```
"""
Modo de juego humano con controles aleatorios - Come Frutas.

Este módulo implementa una versión jugable del entorno donde un humano puede
controlar el agente directamente. La característica única es que los controles
de movimiento se asignan aleatoriamente cada vez que se inicia una partida,
añadiendo un elemento de desafío y adaptabilidad.

Características principales:
- Modo Setup: Configuración manual del escenario
- Modo Humano: Control directo del agente por el jugador
- Controles aleatorios: Mapeo aleatorio de teclas a movimientos
- Interfaz intuitiva: Gráficos y feedback visual
- Desafío adaptativo: Cada partida requiere aprender nuevos controles
```

```
Propósito educativo:
- Comparar rendimiento humano vs. IA
- Experimentar la dificultad de adaptación a controles cambiantes
- Entender la importancia de la consistencia en interfaces
- Apreciar la flexibilidad del aprendizaje humano

Autor: [Tu nombre]
Fecha: Agosto 2025
"""

import pygame
import os
import random
import string

# CONFIGURACIÓN DEL ENTORNO VISUAL
"""
Parámetros visuales y dimensiones de la interfaz de juego.
Utiliza celdas más grandes (120px) para mejor visibilidad durante el juego manual.
"""
GRID_WIDTH = 5              # Ancho de la cuadrícula en celdas
GRID_HEIGHT = 5             # Alto de la cuadrícula en celdas
CELL_SIZE = 120             # Tamaño de cada celda en píxeles (mayor para juego manual)
SCREEN_WIDTH = GRID_WIDTH * CELL_SIZE    # Ancho total de la ventana (600px)
SCREEN_HEIGHT = GRID_HEIGHT * CELL_SIZE  # Alto del área de juego (600px)

# PALETA DE COLORES CONSISTENTE
"""
Esquema de colores oscuro profesional, consistente con otros módulos del proyecto.
"""
COLOR_FONDO = (25, 25, 25)      # Gris muy oscuro para el fondo
COLOR_LINEAS = (40, 40, 40)     # Gris oscuro para líneas de cuadrícula
COLOR_CURSOR = (255, 255, 0)    # Amarillo brillante para cursor de selección
COLOR_TEXTO = (230, 230, 230)   # Gris claro para texto legible

# SISTEMA DE CONTROLES ALEATORIOS
"""
Genera un conjunto de teclas válidas para asignación aleatoria de controles.
Se evitan teclas especiales para prevenir conflictos con funciones del sistema.
"""
TECLAS_VALIDAS = [getattr(pygame, f"K_{c}") for c in string.ascii_lowercase + string.digits]
```

```python
class EntornoHumano:
    """
    Entorno de juego optimizado para control humano directo.

    Esta clase maneja la lógica del juego cuando un humano controla el agente,
    incluyendo movimiento, colisiones, recolección de objetos y condiciones
    de victoria/derrota. Se enfoca en proporcionar feedback inmediato y
    una experiencia de juego fluida.

    Diferencias con entornos de IA:
    - Feedback inmediato con mensajes en consola
    - Lógica de juego simplificada (sin recompensas numéricas)
    - Terminación inmediata en victoria/derrota
    - Controles responsivos para jugabilidad humana

    Attributes:
        agente_pos (tuple): Posición actual del agente (x, y)
        frutas (set): Conjunto de posiciones con frutas
        venenos (set): Conjunto de posiciones con venenos
        paredes (set): Conjunto de posiciones con paredes/obstáculos
    """
    def __init__(self):
        """
        Inicializa el entorno con configuración vacía.

        El agente comienza en la esquina superior izquierda (0,0) y todos
        los conjuntos de elementos están vacíos, permitiendo configuración manual.
        """
        self.agente_pos = (0, 0)    # Posición inicial del agente
        self.frutas = set()         # Conjunto de posiciones de frutas
        self.venenos = set()        # Conjunto de posiciones de venenos
        self.paredes = set()        # Conjunto de posiciones de paredes

    def reset(self):
        """
        Resetea la posición del agente al inicio del juego.

        Coloca al agente en la posición inicial (0,0) sin modificar
        la configuración del escenario. Utilizado al comenzar una nueva partida.
        """
        self.agente_pos = (0, 0)
```

```
    def limpiar(self):
        """
        Elimina todos los elementos del entorno.

        Limpia frutas, venenos y paredes del escenario, dejando una
        cuadrícula vacía para configuración desde cero.
        """
        self.frutas.clear()
        self.venenos.clear()
        self.paredes.clear()

    def step(self, accion):
        """
        Ejecuta una acción del jugador humano en el entorno.

        Procesa el movimiento del agente, verifica colisiones y maneja
        las interacciones con elementos del entorno. Proporciona feedback
        inmediato al jugador mediante mensajes en consola.

        Args:
            accion (int): Dirección de movimiento:
                          0 = Arriba (decrementar y)
                          1 = Abajo (incrementar y)
                          2 = Izquierda (decrementar x)
                          3 = Derecha (incrementar x)

        Returns:
            bool: True si el juego terminó (victoria o derrota), False si continúa
        """
        # Calcular nueva posición basada en la acción
        x, y = self.agente_pos
        if accion == 0:     # Arriba
            y -= 1
        elif accion == 1:   # Abajo
            y += 1
        elif accion == 2:   # Izquierda
            x -= 1
        elif accion == 3:   # Derecha
            x += 1

        # Verificar colisiones: límites del tablero o paredes
        if x < 0 or x >= GRID_WIDTH or y < 0 or y >= GRID_HEIGHT or (x, y) in self.paredes:
```

```
            # Movimiento inválido: no actualizar posición
            return False

        # Movimiento válido: actualizar posición del agente
        self.agente_pos = (x, y)

        # Verificar interacciones con elementos del entorno
        if self.agente_pos in self.frutas:
            # Fruta recogida: eliminar del conjunto
            self.frutas.remove(self.agente_pos)

            # Verificar condición de victoria
            if not self.frutas:
                print("\n ¡Ganaste! Recolectaste todas las frutas.\n")
                return True  # Juego terminado con éxito

        elif self.agente_pos in self.venenos:
            # Veneno tocado: derrota inmediata
            print("\n ¡Oh no! Tocaste un veneno.\n")
            return True  # Juego terminado con fallo

        # Continuar juego
        return False

    def dibujar(self, pantalla, modo, cursor_pos, img_fruta, img_veneno, img_pared, img_agent
        """
        Renderiza el estado completo del entorno con interfaz interactiva.

        Dibuja todos los elementos visuales del juego incluyendo grid, objetos
        del entorno y cursor de selección. Proporciona feedback visual para
        la interacción del jugador en diferentes modos (colocación/juego).

        Args:
            pantalla (pygame.Surface): Superficie donde renderizar
            modo (str): Modo actual de la interfaz ('frutas', 'venenos', 'paredes', 'jugar')
            cursor_pos (tuple): Posición (x,y) del cursor en coordenadas de grid
            img_fruta (pygame.Surface): Sprite de las frutas
            img_veneno (pygame.Surface): Sprite de los venenos
            img_pared (pygame.Surface): Sprite de las paredes
            img_agente (pygame.Surface): Sprite del agente
            _ : Parámetro no utilizado (compatibilidad de interfaz)
```

```
        Note:
            Renderiza en orden específico: fondo, grid, objetos, agente, cursor.
            El cursor cambia de color según el modo de colocación activo.
        """
        # Limpiar pantalla con color de fondo
        pantalla.fill(COLOR_FONDO)

        # Dibujar líneas del grid para guía visual
        for x in range(0, SCREEN_WIDTH, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (x, 0), (x, SCREEN_HEIGHT))
        for y in range(0, SCREEN_HEIGHT, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (0, y), (SCREEN_WIDTH, y))

        # Dibujar elementos del entorno
        for fruta in self.frutas:
            pantalla.blit(img_fruta, (fruta[0]*CELL_SIZE, fruta[1]*CELL_SIZE))
        for veneno in self.venenos:
            pantalla.blit(img_veneno, (veneno[0]*CELL_SIZE, veneno[1]*CELL_SIZE))
        for pared in self.paredes:
            pantalla.blit(img_pared, (pared[0]*CELL_SIZE, pared[1]*CELL_SIZE))

        # Dibujar agente (jugador) - siempre visible en primer plano
        pantalla.blit(img_agente, (self.agente_pos[0]*CELL_SIZE, self.agente_pos[1]*CELL_SIZ

        # Dibujar cursor de selección en modo configuración
        if modo == "SETUP":
            cursor_rect = pygame.Rect(cursor_pos[0]*CELL_SIZE, cursor_pos[1]*CELL_SIZE, CELL_
            pygame.draw.rect(pantalla, COLOR_CURSOR, cursor_rect, 3)

        # Renderizar información de interfaz
        font = pygame.font.Font(None, 30)
        pantalla.blit(font.render(f"Modo: {modo}", True, COLOR_TEXTO), (10, SCREEN_HEIGHT + 5
        pantalla.blit(font.render("F: Fruta, V: Veneno, W: Pared, C: Limpiar, H: Jugar", True
        pantalla.blit(font.render("Descubre los controles ocultos usando letras/números", Tru

def cargar_imagen(nombre, fallback_color):
    """
    Carga una imagen desde archivo con sistema de respaldo.

    Intenta cargar una imagen sprite desde el directorio actual.
    Si la carga falla, crea una superficie de color sólido como respaldo.
    Escala automáticamente al tamaño de celda definido.
```

```
    Args:
        nombre (str): Nombre del archivo de imagen a cargar
        fallback_color (tuple): Color RGB (r,g,b) para superficie de respaldo

    Returns:
        pygame.Surface: Superficie cargada y escalada, o superficie de color
                        si la carga falló

    Note:
        Todas las imágenes se escalan a CELL_SIZE x CELL_SIZE píxeles.
        Utiliza convert_alpha() para optimizar el renderizado con transparencia.
    """
    try:
        # Construir ruta completa al archivo de imagen
        ruta = os.path.join(os.path.dirname(__file__), nombre)
        # Cargar imagen con soporte de transparencia
        img = pygame.image.load(ruta).convert_alpha()
        # Escalar a tamaño de celda estándar
        return pygame.transform.scale(img, (CELL_SIZE, CELL_SIZE))
    except:
        # Crear superficie de respaldo con color sólido si falla la carga
        s = pygame.Surface((CELL_SIZE, CELL_SIZE))
        s.fill(fallback_color)
        return s

def generar_controles_aleatorios():
    """
    Genera un mapeo aleatorio de teclas para controles de movimiento.

    Crea una asignación aleatoria entre teclas del teclado y direcciones
    de movimiento para añadir un elemento de desafío y descubrimiento
    al juego. Los jugadores deben encontrar qué teclas controlan cada dirección.

    Returns:
        dict: Mapeo de códigos de tecla pygame a acciones de movimiento:
              {tecla_pygame: accion_int}
              donde accion_int es 0=Arriba, 1=Abajo, 2=Izquierda, 3=Derecha

    Note:
        Utiliza teclas alfanuméricas (A-Z, 0-9) para máxima compatibilidad.
        Garantiza que cada dirección tenga exactamente una tecla asignada.
    """
```

```
    # Seleccionar 4 teclas aleatorias del conjunto disponible
    teclas = random.sample(TECLAS_VALIDAS, 4)
    # Crear lista de acciones de movimiento
    acciones = [0, 1, 2, 3]  # Arriba, abajo, izquierda, derecha
    # Mezclar aleatoriamente las acciones
    random.shuffle(acciones)
    # Crear diccionario de mapeo tecla->acción
    return dict(zip(teclas, acciones))

def main():
    """
    Función principal del juego en modo humano.

    Inicializa Pygame, configura la ventana de juego y ejecuta el bucle
    principal que maneja dos modos: configuración del entorno y juego
    con controles aleatorios. Proporciona una experiencia interactiva
    donde el jugador puede diseñar niveles y luego jugarlos.

    Flujo del juego:
        1. Modo SETUP: Colocar frutas, venenos y paredes con el mouse
        2. Modo JUGAR: Controlar agente con teclas aleatorias descubiertas
        3. Victoria: Recolectar todas las frutas
        4. Derrota: Tocar veneno

    Controles SETUP:
        - Mouse: Mover cursor
        - F: Colocar fruta
        - V: Colocar veneno
        - W: Colocar pared
        - C: Limpiar todo
        - H: Iniciar juego

    Controles JUGAR:
        - Teclas aleatorias para movimiento (descubrir experimentando)
        - ESC: Volver a configuración
    """
    # Inicializar Pygame y configurar ventana
    pygame.init()
    pantalla = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT + 100))
    pygame.display.set_caption("Modo Humano Aleatorio - Come Frutas")

    # Inicializar entorno y variables de estado
```

```
entorno = EntornoHumano()
cursor_pos = [0, 0]
modo = "SETUP"  # Modo inicial: configuración del entorno
mapeo_controles = {}  # Mapeo de teclas aleatorias (generado al jugar)

# Cargar sprites con colores de respaldo
img_fruta = cargar_imagen("fruta.png", (40, 200, 40))
img_veneno = cargar_imagen("veneno.png", (255, 50, 50))
img_pared = cargar_imagen("pared.jpg", (80, 80, 80))
img_agente = cargar_imagen("agente.png", (60, 100, 255))

# Variables de control del juego
reloj = pygame.time.Clock()
corriendo = True

# Bucle principal del juego
while corriendo:
    # Procesar eventos de entrada
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            corriendo = False
        elif evento.type == pygame.KEYDOWN:
            if evento.key == pygame.K_s:
                modo = "SETUP"
            elif evento.key == pygame.K_h:
                modo = "HUMANO"
                entorno.reset()
                mapeo_controles = generar_controles_aleatorios()

            if modo == "SETUP":
                if evento.key == pygame.K_UP: cursor_pos[1] = max(0, cursor_pos[1]-1)
                elif evento.key == pygame.K_DOWN: cursor_pos[1] = min(GRID_HEIGHT-1, curs
                elif evento.key == pygame.K_LEFT: cursor_pos[0] = max(0, cursor_pos[0]-1
                elif evento.key == pygame.K_RIGHT: cursor_pos[0] = min(GRID_WIDTH-1, curs
                # Colocación de elementos con teclas específicas
                pos = tuple(cursor_pos)
                if evento.key == pygame.K_f:
                    # F: Colocar/quitar fruta (toggle)
                    entorno.frutas.symmetric_difference_update({pos})
                    entorno.venenos.discard(pos)
                    entorno.paredes.discard(pos)
                elif evento.key == pygame.K_v:
```

```
                        # V: Colocar/quitar veneno (toggle)
                        entorno.venenos.symmetric_difference_update({pos})
                        entorno.frutas.discard(pos)
                        entorno.paredes.discard(pos)
                    elif evento.key == pygame.K_w:
                        # W: Colocar/quitar pared (toggle)
                        entorno.paredes.symmetric_difference_update({pos})
                        entorno.frutas.discard(pos)
                        entorno.venenos.discard(pos)
                    elif evento.key == pygame.K_c:
                        # C: Limpiar todo el entorno
                        entorno.limpiar()

                # Controles específicos del modo HUMANO
                elif modo == "HUMANO":
                    if evento.key in mapeo_controles:
                        # Ejecutar acción de movimiento con tecla aleatoria
                        accion = mapeo_controles[evento.key]
                        terminado = entorno.step(accion)
                        if terminado:
                            # Volver a configuración al terminar el juego
                            modo = "SETUP"

        # Renderizar estado actual del juego
        entorno.dibujar(pantalla, modo, cursor_pos, img_fruta, img_veneno, img_pared, img_ag
        pygame.display.flip()
        reloj.tick(30)

    # Limpiar recursos al salir
    pygame.quit()

if __name__ == '__main__':
    main()
```