Proyecto Final IA

Agente Come Frutas

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

1. 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.

2. Desarrollo y Experimentación

El proyecto se desarrolló de forma iterativa, comenzando con un algoritmo fundamental y avanzando hacia técnicas más complejas para superar los desafíos encontrados.

2.1. Punto de Partida: Q-Learning

El desarrollo del agente autónomo inició con la implementación del algoritmo clásico de Aprendizaje por Refuerzo: Q-Learning. Este enfoque permitió construir una base conceptual sólida para comprender cómo un agente puede aprender a actuar en un entorno dinámico sin conocimiento previo del mismo. A través de ensayo y error, el agente aprende una política óptima que maximiza su recompensa acumulada al recolectar frutas y evitar venenos.

Entorno y Representación de Estados

El entorno fue modelado como una cuadrícula de 5×5, donde cada celda puede contener una fruta (recompensa positiva), un veneno (castigo fuerte) o estar vacía. El estado del agente se representa únicamente por su posición (x,y) en la rejilla, lo que resulta en un espacio de estados de tamaño 25 (5 filas × 5 columnas).

```
self.q_table = np.zeros((num_estados[0], num_estados[1], num_acciones))
```

Cada entrada de la tabla Q[s][a] almacena el valor esperado de realizar la acción a en el estado s. El agente puede realizar cuatro acciones posibles: arriba, abajo, izquierda y derecha.

Parámetros del Aprendizaje

Se definieron los siguientes hiperparámetros para controlar el proceso de aprendizaje:

```
# --- 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    # Castigo por movimiento para la eficiencia.
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.
```

- ALPHA define cuánto se actualiza la tabla Q con la nueva información.
- GAMMA determina la importancia del valor futuro frente a la recompensa inmediata.
- EPSILON regula el equilibrio entre exploración y explotación, y su decaimiento permite al agente ser curioso al inicio y luego explotar lo aprendido.

Política de Acción: Epsilon-Greedy

El agente selecciona sus movimientos mediante una política epsilon-greedy:

```
def elegir_accion(self, estado):
    if random.uniform(0, 1) < self.epsilon:
        return random.randint(0, self.num_acciones - 1)  # Exploración
    else:
        return np.argmax(self.q_table[estado])           # Explotación
```

Esta estrategia permite al agente explorar nuevas acciones al inicio del entrenamiento, pero gradualmente se convierte en un experto que toma decisiones óptimas.

Aprendizaje con la Ecuación de Bellman

La función de actualización aplica la ecuación de Bellman, actualizando el valor de la acción ejecutada según la recompensa recibida y el mejor valor futuro estimado:

```
def actualizar_q_table(self, estado, accion, recompensa, nuevo_estado):
    valor_antiguo = self.q_table[estado][accion]
    valor_futuro_maximo = np.max(self.q_table[nuevo_estado])
    nuevo_q = valor_antiguo + ALPHA * (
        recompensa + GAMMA * valor_futuro_maximo - valor_antiguo
    )
    self.q_table[estado][accion] = nuevo_q
```

Este mecanismo es el núcleo del aprendizaje en Q-Learning: el agente se adapta en función de la retroalimentación recibida del entorno.

Entrenamiento del Agente

El agente se entrena a lo largo de 20,000 episodios, en los que explora y aprende sobre el entorno. Cada episodio se reinicia desde la posición inicial y termina cuando el agente recolecta todas las frutas o toca un veneno.

```
for episodio in range(NUM_EPISODIOS_ENTRENAMIENTO):
    entorno.frutas = frutas_iniciales.copy()
    entorno.venenos = venenos_iniciales.copy()
    estado = entorno.reset_a_configuracion_inicial()
    terminado = False
    while not terminado:
        accion = agente.elegir_accion(estado)
        nuevo_estado, recompensa, terminado = entorno.step(accion, "TRAINING")
        agente.actualizar_q_table(estado, accion, recompensa, nuevo_estado)
        estado = nuevo_estado
    agente.decaimiento_epsilon()
```

Visualización y Modo Juego

Una vez entrenado, el agente puede ser probado en el modo PLAYING, donde actúa usando únicamente explotación (=0). El entorno fue diseñado con pygame, incluyendo sprites para representar frutas, venenos y paredes, lo cual facilita visualizar las decisiones del agente en tiempo real.

Limitaciones Observadas

Aunque este enfoque demostró ser funcional en escenarios simples, presentó problemas de convergencia y eficiencia cuando el número de frutas y venenos aumentaba. La tabla Q escala mal en entornos con más dimensiones, lo que motivó la transición hacia métodos más avanzados como Deep Q-Networks.

2.2 DQN con CNN (Deep Q-Network con Red Convolucional)

Motivación y Limitaciones del Q-Learning

Tras observar las limitaciones del Q-Learning clásico —en particular su incapacidad para escalar a entornos complejos o generalizar eficientemente— se adoptó el algoritmo Deep Q-Network (DQN) como siguiente paso evolutivo. Este enfoque reemplaza la tabla Q discreta por una red neuronal convolucional (CNN), capaz de aproximar los valores Q mediante aprendizaje profundo a partir de representaciones visuales del entorno.

Representación del Estado como Imagen Multicanal

Cada estado del entorno fue transformado en un tensor tridimensional de forma (3, 5, 5), donde:

- Canal 0 indica la posición del agente.
- Canal 1 marca las frutas.
- Canal 2 representa los venenos.

Esta representación permite que la CNN procese el entorno como una "imagen", reconociendo patrones espaciales.

```
# environment.py
state = np.zeros((3, self.size, self.size), dtype=np.float32)
state[0, agent_pos[0], agent_pos[1]] = 1.0  # Canal del agente
state[1, fruta[0], fruta[1]] = 1.0          # Canal de frutas
state[2, veneno[0], veneno[1]] = 1.0        # Canal de venenos
```

Arquitectura de la Red Neuronal

La red CNN_DQN, definida en agent.py, sigue una arquitectura sencilla pero eficaz:

- 2 capas convolucionales para extraer características espaciales.
- 2 capas densas para calcular el valor Q de cada acción

```
# Arquitectura de la red neuronal que actúa como el "cerebro" visual del agente.
class CNN_DQN(nn.Module):
    def __init__(self, h, w, outputs): [cite: 1279]
        super(CNN_DQN, self).__init__()
        # Capas convolucionales para "ver" el tablero
        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 del tamaño para la capa lineal
        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 lineales para "decidir" el valor de cada acción
        self.fc1 = nn.Linear(linear_input_size, 256)
        self.fc2 = nn.Linear(256, outputs)

    def forward(self, x):
        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)
```

Esta red recibe como entrada el estado en forma de tensor y devuelve un vector de valores Q (uno por acción).

Componentes del Agente DQN

El agente implementa técnicas avanzadas del algoritmo DQN según la arquitectura de Mnih et al. (2015):

a) Red principal y red objetivo

- La red principal es la que se entrena activamente.
- La red objetivo se actualiza cada cierto número de pasos (update_target_every) para estabilizar el entrenamiento.

```
self.target_model.load_state_dict(self.model.state_dict())
```

b) Memoria de Replay

El agente almacena transiciones pasadas en una cola circular (deque), desde donde extrae mini-batches para entrenamiento aleatorio. Esto rompe la correlación temporal entre muestras y mejora la estabilidad.

```
self.memory = deque(maxlen=20000)
self.memory.append((state, action, reward, next_state, done))
```

c) Política Epsilon-Greedy

Durante el entrenamiento, el agente decide sus acciones con una política que balancea exploración y explotación. Con probabilidad , elige una acción aleatoria; de lo contrario, toma la acción con mayor valor Q:

```
if explore and np.random.rand() <= self.epsilon:
    return random.randrange(self.action_size)
...
action_values = self.model(state_tensor)
return np.argmax(action_values.cpu().data.numpy())
```

El valor de decae progresivamente para favorecer la explotación del conocimiento adquirido:

```
if self.epsilon > self.epsilon_min:
    self.epsilon *= self.epsilon_decay
```

Proceso de Entrenamiento

El agente entrena su red principal mediante retropropagación en lotes aleatorios (batch_size) tomados de la memoria de replay. Cada lote actualiza la red usando la pérdida entre los valores actuales y los objetivos (targets):

```
# Cálculo del valor objetivo (target Q)
target_q_values = rewards + (gamma * next_q_values * (~dones))
loss = criterion(current_q_values, target_q_values)
loss.backward()
optimizer.step()
```

El uso de torch.no_grad() para calcular los targets evita que se acumulen gradientes en la red objetivo.

Interfaz Visual e Interacción

El entorno visual fue implementado en main.py usando Pygame. Incluye dos modos:

- SETUP: permite al usuario colocar frutas y venenos manualmente en el tablero.
- RUN: el agente entrenado toma el control y ejecuta su política aprendida, visualizando su comportamiento en tiempo real.

Esto permitió una validación cualitativa del rendimiento del agente en diferentes configuraciones.

Resultados y Observaciones

- El uso de CNN permitió al agente generalizar mejor en diferentes escenarios.
- El sistema de reward shaping (recompensas por acercarse a frutas) mejoró la eficiencia del entrenamiento.
- La representación como imagen multicanal ayudó a la red a reconocer patrones espaciales útiles para la navegación.

2.3 DDQN

Problemas con DQN clásico

Aunque Deep Q-Networks (DQN) mejoró la generalización del agente frente al Q-Learning tradicional, aún sufría de un problema conocido como overestimation bias: la tendencia a sobrevalorar las recompensas futuras. Esto ocurre porque DQN utiliza la misma red para seleccionar y evaluar las acciones futuras, lo que puede introducir errores acumulativos.

Para mitigar este problema, se implementó el algoritmo Double DQN (DDQN), una mejora propuesta por Van Hasselt et al. (2016), que desacopla el proceso de selección de acción del de evaluación de valor. Esto reduce el sesgo y mejora la estabilidad del entrenamiento.

Diferencia Clave: Selección vs Evaluación

En DQN clásico:

```
target = reward + gamma * max(Q_target(next_state))
```

En Double DQN:

```
# Selección con red principal
next_actions = model(next_state).argmax(dim=1)

# Evaluación con red objetivo
next_q = target_model(next_state).gather(1, next_actions.unsqueeze(1))
target = reward + gamma * next_q
```

Este cambio sutil mejora notablemente la precisión de la estimación de los valores Q futuros.

Implementación en el Proyecto

La clase Agent en agent.py fue modificada para aplicar el criterio Double DQN en el método replay(). La arquitectura de la red se mantuvo similar al agente DQN, pero se ajustó la forma en que se calcula el valor objetivo durante el entrenamiento.

Fragmento clave del método replay:

```
# 1. Selección con la red principal
next_actions = self.model(next_states).detach().argmax(dim=1).unsqueeze(1)

# 2. Evaluación con la red objetivo
next_q_values = self.target_model(next_states).gather(1, next_actions)

# 3. Cálculo de target
targets = rewards + gamma * next_q_values * (1 - dones)
```

Otras Mejoras y Técnicas Usadas

Además del cambio en el cálculo de los valores objetivos, el agente DDQN mantiene:

- Una política epsilon-greedy con decaimiento gradual.
- Replay Buffer circular para almacenar transiciones de entrenamiento.

- Actualización periódica de la red objetivo para estabilizar el aprendizaje.
- Entrenamiento mini-batch con retropropagación en train.py.

Interfaz Interactiva

La demostración del agente DDQN se integró a través de main.py, mientras que el entrenamiento puede ser gestionado desde interfaztrain.py. La representación visual de frutas, venenos y el agente permite observar cómo la política aprendida se adapta al entorno.

Resultados Obtenidos

Durante el entrenamiento de DDQN, se observó una mejora significativa en:

- Estabilidad del aprendizaje: menos oscilaciones durante el entrenamiento.
- Precisión en la toma de decisiones: el agente evitó más venenos sin necesidad de aumentar la exploración.
- Generalización: el agente entrenado pudo actuar correctamente en escenarios no vistos durante el entrenamiento.

2.4. Paradigma Alternativo: Algoritmos Genéticos

Motivación

A diferencia de los algoritmos anteriores que dependen de gradientes y retropropagación (como Q-Learning o DQN), el enfoque evolutivo busca optimizar directamente las políticas de los agentes mediante la simulación de un proceso de selección natural. Esta técnica resulta especialmente útil cuando se trabaja con entornos ruidosos o no diferenciables, donde las funciones de pérdida tradicionales pueden fallar o llevar a soluciones subóptimas.

Fundamentos del Algoritmo Genético

El algoritmo genético implementado se basa en los principios fundamentales de la evolución biológica:

1. Inicialización: Se genera una población aleatoria de agentes, cada uno con una red neuronal que actúa como su "ADN".
2. Evaluación (Fitness): Cada agente se enfrenta a un entorno dinámico y su desempeño se mide como la suma de recompensas obtenidas.
3. Selección: Se eligen los agentes con mayor fitness para reproducirse.
4. Cruzamiento (Crossover): Los genes de dos padres se combinan para formar un nuevo agente hijo.
5. Mutación: Se introduce variación aleatoria en los genes del hijo.

6. Elitismo y Reemplazo: Se conservan los mejores agentes y se forma una nueva generación con los hijos generados.

Este ciclo se repite durante múltiples generaciones hasta que el agente evoluciona una política óptima.

Implementación del Agente Genético

La arquitectura general del entrenamiento se encuentra en train_ga.py, y el comportamiento del agente está definido en agent_ga.py.

a) Estructura de la Población

Cada agente está representado por una red neuronal densa que recibe el estado del entorno y genera una acción discreta. Los pesos de esta red constituyen su genotipo. Inicialmente, los pesos son asignados aleatoriamente:

```
def create_initial_population():
    return [Agent() for _ in range(POPULATION_SIZE)]
```

b) Evaluación de Fitness

Cada agente es evaluado en un escenario aleatorio donde debe recolectar frutas y evitar venenos. Su desempeño se mide por la suma total de recompensas:

```
for _ in range(50):
    action = agent.choose_action(state)
    state, reward, done = env.step(action)
    total_reward += reward
```

Este total es almacenado como su fitness, y determina su posibilidad de ser seleccionado como padre.

c) Selección de Padres

Se selecciona el 20% de los mejores agentes de la población como candidatos para reproducción:

```
population.sort(key=lambda x: x.fitness, reverse=True)
parents = population[:int(POPULATION_SIZE * 0.2)]
```

d) Cruzamiento Uniforme

La recombinación de genes se realiza de manera uniforme: cada peso tiene una probabilidad del 50% de heredarse de cada padre:

```
for key in child_genes.keys():
    if random.random() < 0.5:
        child_genes[key] = p1_genes[key].clone()
    else:
        child_genes[key] = p2_genes[key].clone()
```

e) Mutación Gaussiana

Cada parámetro tiene una probabilidad de ser alterado añadiendo ruido gaussiano con desviación estándar 0.1, lo que permite explorar nuevas soluciones:

```
noise = torch.randn_like(child_genes[key]) * 0.1
child_genes[key] += noise
```

2.5 Solución Definitiva: Aprendizaje por Imitación y Currículo

El aprendizaje por imitación (Imitation Learning) es una técnica del aprendizaje automático en la cual un agente aprende a comportarse observando las decisiones de un experto. En lugar de aprender únicamente a través de la exploración y la retroalimentación de recompensas como en Q-learning o DQN, el agente intenta replicar el comportamiento observado, generalizando a partir de ejemplos ya resueltos.

Este enfoque es útil en tareas donde una política experta está disponible o es más eficiente que explorar de forma aleatoria. El agente no necesita experimentar con acciones riesgosas o costosas, sino que puede aprender directamente de ejemplos correctos.

Arquitectura General

En este proyecto, el agente utiliza una red neuronal convolucional para mapear directamente una representación visual del entorno (estado) hacia una acción (arriba, abajo, izquierda o derecha). El entorno es una cuadrícula de 5x5 donde el agente debe aprender a alcanzar frutas y evitar venenos.

La red se entrena como un clasificador supervisado, usando los datos generados por un experto planificador (algoritmo A*) que provee las acciones correctas para diferentes configuraciones del entorno.

Generación de Datos Expertos

La clase AStarSolver ubicada en a_star_solver.py es responsable de encontrar la ruta óptima entre el agente y las frutas. El archivo generate_data.py genera un conjunto de ejemplos estado-acción (dataset) mediante simulaciones automáticas con A*.

Cada entrada del dataset contiene:

- Un estado del entorno como tensor 3x5x5 (canales para frutas, venenos y posición del agente).
- La acción óptima sugerida por A* en ese escenario.

Entrenamiento con Curriculum Learning

El entrenamiento se realiza mediante el archivo train_curriculum.py, que implementa un enfoque Curriculum Learning. Este método consiste en enseñar al agente desde situaciones simples hasta escenarios más complejos, de forma progresiva, para mejorar la estabilidad y la capacidad de generalización.

El currículo consta de 4 niveles:

Lección	Nº Frutas	Dataset	Nº Épocas
1	1	`expert_data_1_fruit.pkl`	25
2	2	`expert_data_2_fruits.pkl`	30
3	3	`expert_data_3_fruits.pkl`	40
4	4	`expert_data_4_fruits.pkl`	50

Durante el entrenamiento, el modelo aprende a clasificar correctamente la mejor acción en cada estado. Se usa la función CrossEntropyLoss y el optimizador Adam.

Fragmento clave del entrenamiento:

```
states = torch.FloatTensor(np.array([item[0] for item in data]))   # Entradas
actions = torch.LongTensor(np.array([item[1] for item in data]))   # Etiquetas

outputs = model(states)                    # Inferencia
loss = criterion(outputs, actions)         # Cálculo de pérdida
loss.backward()                            # Retropropagación
optimizer.step()                           # Actualización de pesos
```

Demostración del Agente

Una vez entrenado, el modelo se guarda como imitacion_model.pth y se utiliza en el archivo main.py para la simulación interactiva. Esta interfaz permite al usuario:

- Configurar frutas y venenos con el mouse.
- Presionar la barra espaciadora para que el agente actúe automáticamente.
- Visualizar el movimiento paso a paso del agente basado en inferencia con la red entrenada.

Fragmento del ciclo principal en main.py:

```
state = env.get_state()
action = agent.choose_action(state)
_, _, done = env.step(action)
```

Resultados Observados

- Rápida convergencia: El agente aprende en pocos minutos gracias a la guía del experto.
- Comportamiento estable: Sigue rutas seguras hacia las frutas.
- Alta precisión en escenarios nuevos, especialmente con 1 o 2 frutas.

Sin embargo, en escenarios con múltiples frutas o venenos muy cercanos, puede cometer errores, lo cual indica una limitación del entrenamiento supervisado frente a la necesidad de adaptación en entornos dinámicos.

2.6 Jugador humano

Como parte del proceso de evaluación comparativa, se implementó una versión del entorno que permite la interacción directa de un ser humano. Este enfoque no se basa en técnicas de aprendizaje automático, sino en la observación y toma de decisiones por parte de un usuario humano. Esta modalidad tiene dos propósitos principales:

1. Establecer una línea base humana para comparar el rendimiento de los agentes entrenados.
2. Permitir una evaluación cualitativa de la dificultad del entorno desde la perspectiva de un jugador humano.

Implementación

La implementación está contenida en el archivo Humano.py, que define una interfaz visual e interactiva con el entorno. Se utiliza la biblioteca pygame para permitir el control del agente mediante el teclado. El usuario puede mover al agente en tiempo real con las teclas de dirección y observar el entorno completo.

```
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_UP:
            env.move_agent(0, -1)
        elif event.key == pygame.K_DOWN:
            env.move_agent(0, 1)
        elif event.key == pygame.K_LEFT:
```

```
            env.move_agent(-1, 0)
        elif event.key == pygame.K_RIGHT:
            env.move_agent(1, 0)
```

Cada vez que el jugador presiona una tecla:

- El agente se mueve en la dirección correspondiente.
- El entorno se actualiza gráficamente.
- Se calcula si ha recolectado una fruta o caído en veneno.

También se muestra en pantalla el puntaje actual, el número de frutas recolectadas y si el juego ha terminado.

Experiencia de Usuario

Durante la ejecución, se observó que el control humano resulta efectivo en escenarios simples, pero tiende a:

- Perder eficiencia en situaciones con múltiples venenos.
- Reaccionar más lentamente que los agentes automatizados entrenados.
- Cometer errores de juicio, especialmente cuando la ubicación de los objetos requiere planificación anticipada.

Estas observaciones refuerzan la importancia de los algoritmos de planificación y aprendizaje, ya que el ser humano suele actuar de forma reactiva más que estratégica.

Comparación y Valor

Aunque el agente humano no constituye una estrategia de inteligencia artificial, su desempeño sirve como punto de referencia:

- Permite validar si el entorno es razonablemente jugable.
- Ayuda a calibrar la dificultad de los escenarios utilizados en el entrenamiento de los agentes.
- Ofrece datos reales para diseñar modelos de imitación, en caso de querer capturar acciones humanas como dataset.

2.7 Estructura del Entorno: Variaciones según el tipo de agente

Durante el desarrollo del proyecto se utilizaron múltiples versiones del archivo environment.py, cada una adaptada a las necesidades particulares del enfoque de aprendizaje o control implementado. Si bien todas las versiones representan un entorno común basado en una cuadrícula de 5×5 con frutas, venenos y un agente, su estructura interna, funcionalidad y nivel de abstracción varían considerablemente dependiendo del tipo de agente que lo utiliza.

Esta modularidad permitió adaptar cada entorno a los requerimientos de cada técnica, optimizando su compatibilidad y rendimiento. A continuación, se presenta una descripción detallada de las diferencias clave entre los entornos utilizados:

1. Entorno para DQN y DDQN

- Propósito: Aprendizaje profundo mediante redes neuronales convolucionales.
- Formato de estado: Devuelve un tensor 3D de dimensiones (3, 5, 5) que representa en canales separados al agente, frutas y venenos. Este formato es compatible con entradas de CNN.
- Método step(): Implementa transiciones completas: toma una acción, actualiza el entorno, calcula la recompensa y verifica la condición de finalización.
- Recompensas: Dinámicamente calculadas en función del objeto con el que interactúa el agente (fruta, veneno o celda vacía).
- Visualización: No tiene interfaz gráfica integrada, ya que se utiliza principalmente para entrenamiento automático.

2. Entorno para Algoritmo Genético

- Propósito: Evaluación de agentes generados por algoritmos evolutivos.
- Formato de estado: No se genera un estado tensorial ni se almacena como entrada; se manipula directamente en el entrenamiento.
- Método step(): Su implementación es simplificada y no retorna recompensas ni estados, sino que actualiza posiciones y retorna si el juego ha finalizado.
- Recompensas: Se evalúan fuera del entorno mediante una función de fitness personalizada en train_ga.py.
- Visualización: No incluye renderizado gráfico ni visualización textual.

3. Entorno para Imitación

- Propósito: Generación de datos supervisados a partir de trayectorias óptimas.
- Formato de estado: Genera vectores de entrada compatibles con aprendizaje supervisado. Se serializa el estado como una matriz procesable por un clasificador.

- Método step(): Calcula transición y recompensa, pero también permite generar datasets estado-acción.
- Recompensas: Personalizadas para facilitar aprendizaje por imitación.
- Visualización: Puede renderizarse en consola o guardar datos sin visualización.

4. Entorno para Agente Humano

- Propósito: Permitir a un usuario jugar manualmente mediante teclado.
- Formato de estado: No expone estados formales ni vectores; el entorno responde directamente a las teclas presionadas.
- Método step(): No se implementa como función explícita; se controla en tiempo real con eventos de teclado.
- Recompensas: Se calculan internamente, pero no se registran para entrenamiento.
- Visualización: Gráfica, mediante la librería pygame. El jugador ve el entorno completo en pantalla.

2.8 Comparativa de Rendimiento entre Agentes

Durante el desarrollo del proyecto, se implementaron y entrenaron distintos tipos de agentes para resolver el entorno "Come Frutas". Cada agente utilizó una técnica de aprendizaje diferente: Q-Learning, DQN con CNN, DDQN, aprendizaje por imitación, evolución genética y control humano. A continuación se presenta un análisis comparativo de su desempeño, tanto desde el punto de vista cualitativo como conductual.

- **Comportamiento Observado**

Agente	Descripción del Comportamiento Observado
Q-Learning	Aprende a jugar, pero su progreso es lento. En niveles complejos necesita más entrenamiento del permitido.
DQN (con CNN)	Juega de forma competente incluso en dificultades altas, aunque ignora la última fruta en algunos casos.
DDQN (imitación con red)	Funciona al inicio, pero tras recolectar una fruta se desorienta completamente y no sigue con el objetivo.
Agente Genético	No juega activamente; su estrategia aprendida es quedarse quieto para evitar penalizaciones.
Imitación	Juega bien en niveles de baja y media dificultad, pero falla al generalizar cuando el entorno se complica.
Humano	No incluido en este análisis automático, ya que el enfoque se centra en agentes computacionales.

- **Análisis Cualitativo**

Q-Learning mostró una curva de aprendizaje lenta. Aunque llega a recolectar frutas en entornos simples, en niveles más exigentes no logra optimizar su comportamiento. Esto se debe en parte a la limitación de episodios y a su exploración basada en tablas que no escalan bien.

DQN (con CNN) logró los mejores resultados en términos de generalización. Su uso de convoluciones permite abstraer mejor el estado del entorno. Sin embargo, su comportamiento al recolectar todas las frutas no es óptimo, ya que tiende a ignorar la última o a quedarse en bucles.

DDQN buscaba mejorar la estabilidad del aprendizaje con redes separadas para el valor objetivo, pero en este caso resultó menos eficiente: tras recolectar una fruta, el agente parece perder el rumbo, lo cual sugiere una mala generalización en los valores futuros.

El agente genético, en lugar de aprender a recolectar frutas, desarrolló una "estrategia segura" que consiste en quedarse quieto, lo que evita penalizaciones por veneno, pero también limita la recompensa. Es un ejemplo claro de cómo la función de fitness puede inducir políticas subóptimas si no está bien diseñada.

El agente por imitación, entrenado con trayectorias óptimas generadas por A*, logró un comportamiento muy competente en mapas simples. Sin embargo, su rendimiento cae cuando se enfrenta a escenarios con estructuras distintas a las observadas en el conjunto de entrenamiento.

3. Resultados y Conclusión

El viaje a través de estos diferentes paradigmas de IA fue revelador. Mientras que los enfoques de Aprendizaje por Refuerzo y Algoritmos Genéticos mostraron potencial, también exhibieron dificultades para converger de manera consistente en un entorno tan "frágil", donde un solo error puede llevar al fracaso.

La estrategia de Aprendizaje por Imitación combinada con el Aprendizaje por Currículo demostró ser la más efectiva y robusta. Al aprender de un experto perfecto y hacerlo de manera gradual, el agente final fue capaz de generalizar su conocimiento y resolver de manera confiable tanto escenarios simples como complejos. Para la demostración visual, se desarrolló una interfaz en Pygame que permite a los usuarios configurar sus propios niveles y observar al agente entrenado resolverlos en tiempo real.

Este proyecto subraya una lección fundamental en el desarrollo de IA: a menudo, la estrategia de entrenamiento y la calidad de los datos son tan o más importantes que la elección del algoritmo en sí. El resultado es un agente competente y una profunda comprensión práctica de los desafíos y soluciones en el campo del Machine Learning.