UNIVERSIDAD MAYOR REAL Y PONTIFICIA DE SAN FRANCISCO XAVIER DE CHUQUISACA
FACULTAD DE CIENCIAS Y TECNOLOGÍA

Estudiante: Medrano Medrano Adrian Mijael

Carrera: Ingeniería de Sistemas

Materia: SIS420

Hockey 2D: Aprendizaje por Refuerzo Multi-Agente con Q-learning Independiente

Índice

---

1. Introducción

1.1 Descripción del Problema

Este proyecto implementa un entorno de **Hockey 2D** donde dos agentes autónomos aprenden a jugar mediante **Aprendizaje por Refuerzo (Reinforcement Learning)**. El objetivo es que cada agente desarrolle una estrategia para marcar goles mientras defiende su propia portería, todo esto sin programación explícita de reglas de juego.

1.2 Objetivos

- **Objetivo General:** Desarrollar e implementar un sistema multi-agente donde dos jugadores aprenden a jugar hockey mediante Q-Learning Independiente.
- **Objetivos Específicos:**
    - Diseñar un entorno discretizado de hockey 2D
    - Implementar el algoritmo Q-Learning para cada agente
    - Visualizar el proceso de aprendizaje y las partidas
    - Analizar la convergencia y efectividad del aprendizaje

1.3 Metodología

Se utiliza **Q-Learning Independiente**, donde cada agente mantiene su propia tabla Q y aprende su política de forma autónoma, sin conocimiento explícito de las políticas de los demás agentes. El entorno se simula mediante programación orientada a objetos en Python, utilizando Pygame para la visualización.

---

2. Marco Teórico

2.1 Aprendizaje por Refuerzo

El **Aprendizaje por Refuerzo (RL)** es un paradigma de aprendizaje automático donde un agente aprende a tomar decisiones mediante interacción con un entorno. Los componentes fundamentales son:

Componentes Básicos

- **Agente:** Entidad que toma decisiones (los jugadores de hockey)
- **Entorno:** El mundo con el que interactúa el agente (la cancha de hockey)
- **Estado (s):** Representación de la situación actual del entorno
- **Acción (a):** Decisión que toma el agente en un estado dado
- **Recompensa (r):** Señal numérica que indica qué tan buena fue una acción
- **Política (π):** Estrategia que mapea estados a acciones

Ciclo de Interacción

Agente

↓ (acción)

Entorno

↓ (estado, recompensa)

Agente

El agente observa el estado, ejecuta una acción, recibe una recompensa y un nuevo estado, y repite el proceso aprendiendo de la experiencia.

2.2 Q-Learning

Q-Learning es un algoritmo de aprendizaje por refuerzo que aprende el valor óptimo de cada par estado-acción.

Función Q

La función Q(s, a) representa el valor esperado de tomar la acción **a** en el estado **s** y seguir la política óptima después:

Q(s, a) = valor esperado de recompensa acumulada

Ecuación de Actualización

Q(s, a) ← Q(s, a) + α[r + γ max_a' Q(s', a') - Q(s, a)]

Donde:

- **α (alpha):** Tasa de aprendizaje (0.2 en nuestro caso)
    - Controla cuánto cambia Q en cada actualización
- **γ (gamma):** Factor de descuento (0.95 en nuestro caso)
    - Importancia de recompensas futuras vs inmediatas
    - Valores cercanos a 1: visión a largo plazo
- **r:** Recompensa inmediata recibida
- **s':** Nuevo estado después de ejecutar la acción
- **max_a' Q(s', a'):** Mejor valor Q posible en el siguiente estado

Política ε-Greedy

Para balancear **exploración** (probar nuevas acciones) y **explotación** (usar conocimiento actual), se usa una política ε-greedy:

Con probabilidad ε: elegir acción aleatoria (exploración)

Con probabilidad (1-ε): elegir mejor acción conocida (explotación)

El valor de ε decrece con el tiempo:

- **ε inicial:** 0.9 (mucha exploración al inicio)
- **ε mínimo:** 0.05 (siempre algo de exploración)
- **Decaimiento:** 0.995 por episodio

2.3 Aprendizaje Multi-Agente

En escenarios multi-agente, múltiples agentes aprenden simultáneamente en un entorno compartido. Esto introduce complejidad adicional:

Q-Learning Independiente

En este enfoque:

- Cada agente mantiene su propia tabla Q
- Cada agente aprende sin considerar explícitamente a los demás
- El otro agente es tratado como parte del entorno
- **Ventaja:** Simplicidad de implementación
- **Desventaja:** No garantiza convergencia al equilibrio óptimo

---

3. Diseño del Entorno

3.1 Configuración del Mundo

Dimensiones de la Cancha

Python

```
N_ROWS, N_COLS = 6, 7

N_ROWS = 6  # 6 filas

N_COLS = 7  # 7 columnas
```

La cancha es una cuadrícula discreta de 6×7 celdas, lo que simplifica el espacio de estados y permite una representación visual clara.

Elementos del Juego

1. **Jugador 1 (Azul):** Inicia en la parte inferior (fila 5, columna 3)
2. **Jugador 2 (Naranja):** Inicia en la parte superior (fila 0, columna 3)
3. **Puck (Blanco):** Inicia cerca del jugador superior (fila 1, columna 3)

Arcos (Porterías)

Python

```
GOAL_WIDTH_CELLS = 3  # Ancho del arco: 3 celdas

GOAL_CENTER_COL = 3   # Centrado en la columna 3
```

- **Arco superior (verde):** Meta del Jugador 1
- **Arco inferior (rojo):** Meta del Jugador 2

Los arcos están centrados y ocupan las columnas 2, 3 y 4.

3.2 Acciones Disponibles

Cada jugador puede ejecutar 4 acciones por turno:

Python

```
A_UP, A_DOWN, A_LEFT, A_RIGHT = range(4)
ACTIONS = [A_UP, A_DOWN, A_LEFT, A_RIGHT]

A_UP = 0    # Mover hacia arriba

A_DOWN = 1  # Mover hacia abajo

A_LEFT = 2  # Mover hacia la izquierda

A_RIGHT = 3 # Mover hacia la derecha
```

Los movimientos están limitados por los bordes de la cancha (los jugadores no pueden salirse).

3.3 Representación del Estado

Estado Completo

El estado global del juego contiene 7 componentes:

python

```
s = (ar1, ac1, ar2, ac2, pr, pc, pd)
```

Donde:

- **ar1, ac1:** Fila y columna del Jugador 1
- **ar2, ac2:** Fila y columna del Jugador 2
- **pr, pc:** Fila y columna del Puck
- **pd:** Dirección vertical del puck (+1 baja, -1 sube)

Estado Codificado

Para reducir el espacio de estados, cada agente observa una versión **codificada** del estado basada en **diferencias relativas**:

python

```
# Para Jugador 1:
dr = clip(pr - ar1, -2, 2)  # Diferencia vertical con el puck
dc = clip(pc - ac1, -2, 2)  # Diferencia horizontal con el puck
estado_J1 = (dr, dc, pd)
```

Ventajas:

- Reduce significativamente el espacio de estados
- Captura la información más relevante (posición relativa al puck)
- Hace el aprendizaje más eficiente

Ejemplo:

- Si el puck está 1 celda arriba y 2 a la derecha: (-1, 2, +1)
- Esto le indica al agente hacia dónde moverse

3.4 Dinámica del Puck

El puck no es controlado directamente por los jugadores, sino que sigue reglas específicas:

Movimiento Vertical

python

```
pr = pr + pd  # Se mueve en la dirección actual
```

El puck se mueve una celda verticalmente en cada turno según su dirección (pd).

Drift Lateral (Movimiento Horizontal Aleatorio)

python

```
def lateral_drift(self, pc):
    r = random.random()
    step = -1 if r < 0.15 else (1 if r < 0.30 else 0)
    # 15% probabilidad de ir izquierda
    # 15% probabilidad de ir derecha
    # 70% probabilidad de no moverse lateralmente
```

Este movimiento aleatorio simula la imprevisibilidad de un puck real.

Rebote al Golpear

Cuando un jugador ocupa la misma celda que el puck:

python

```
if (ar1, ac1) == (pr, pc):  # Jugador 1 golpea
    pd = -pd  # Invierte dirección vertical
```

Esto permite a los jugadores devolver el puck hacia el arco contrario.

3.5 Sistema de Recompensas

El sistema de recompensas guía el aprendizaje de los agentes:

Recompensas por Paso

python

```
STEP_PENALTY = -0.1
```

Cada turno tiene un costo de -0.1 para ambos jugadores, incentivando terminar partidas rápidamente.

Recompensas por Gol

python

```
# Si el puck entra al arco superior:
r1 = +1.0   # Jugador 1 gana
r2 = -1.0   # Jugador 2 pierde

# Si el puck entra al arco inferior:
r1 = -1.0   # Jugador 1 pierde
r2 = +1.0   # Jugador 2 gana
```

Condiciones de Gol

Un gol ocurre cuando:

1. El puck alcanza la fila del arco (0 o 5)
2. El puck está dentro del rango horizontal del arco (columnas 2-4)
3. El puck se mueve en la dirección correcta (entrando al arco)

python

```
in_top_goal = (pr == 0) and (2 <= pc <= 4) and (pd == -1)
in_bottom_goal = (pr == 5) and (2 <= pc <= 4) and (pd == +1)
```

3.6 Visualización con Pygame

Dimensiones de la Ventana

python

```
CELL = 64     # 64 píxeles por celda
MARGIN = 40   # 40 píxeles de margen
W = 7*64 + 2*40 = 528  # Ancho total
H = 6*64 + 2*40 = 464  # Alto total
```

Colores

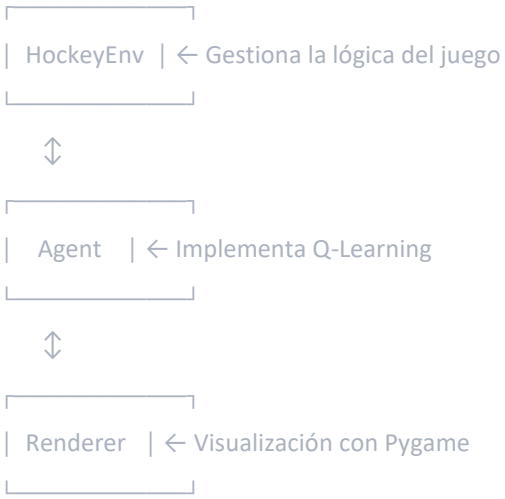
- **Fondo:** Gris oscuro (25, 28, 35)
- **Grilla:** Gris claro (70, 70, 80)
- **Jugador 1:** Azul (70, 170, 255)
- **Jugador 2:** Naranja (255, 170, 70)
- **Puck:** Blanco (240, 240, 240)
- **Arco superior:** Verde (50, 160, 80)
- **Arco inferior:** Rojo (200, 70, 70)

---

4. Implementación

4.1 Arquitectura del Sistema

El código está estructurado en tres clases principales siguiendo el paradigma de **Programación Orientada a Objetos (POO)**:



4.2 Clase HockeyEnv

Esta clase maneja toda la lógica del juego.

Atributos Principales

python

```python
class HockeyEnv:
    STEP_PENALTY = -0.1

    def __init__(self):
        self.ar1, self.ac1 = START_AGENT1  # Posición Jugador 1
        self.ar2, self.ac2 = START_AGENT2  # Posición Jugador 2
        self.pr, self.pc = START_PUCK      # Posición Puck
        self.pd = START_DIR                # Dirección Puck
```

Métodos Clave

reset(): Reinicia el juego a las posiciones iniciales

python

```
def reset(self):
    self.ar1, self.ac1 = START_AGENT1
    self.ar2, self.ac2 = START_AGENT2
    self.pr, self.pc = START_PUCK
    self.pd = START_DIR
    return self.get_state()
```

move(rc, action): Mueve un jugador según la acción

python

```
def move(self, rc, action):
    r, c = rc
    if action == A_UP:    r -= 1
    if action == A_DOWN:  r += 1
    if action == A_LEFT:  c -= 1
    if action == A_RIGHT: c += 1
    # Limitar a los bordes
    r = max(0, min(N_ROWS-1, r))
    c = max(0, min(N_COLS-1, c))
    return (r, c)
```

step(a1, a2): Ejecuta un turno completo

python

```
def step(self, a1, a2):
    # 1. Mover jugadores
    self.ar1, self.ac1 = self.move((self.ar1, self.ac1), a1)
    self.ar2, self.ac2 = self.move((self.ar2, self.ac2), a2)

    # 2. Mover puck
    self.pc = self.lateral_drift(self.pc)
    self.pr = self.pr + self.pd

    # 3. Detectar golpes
    hit1 = (self.ar1, self.ac1) == (self.pr, self.pc)
    hit2 = (self.ar2, self.ac2) == (self.pr, self.pc)
```

```
    if hit1 or hit2:
        self.pd = -self.pd  # Rebote

    # 4. Calcular recompensas
    r1 = r2 = self.STEP_PENALTY
    done = False

    # 5. Detectar goles
    if in_top_goal:
        r1, r2, done = +1.0, -1.0, True
    elif in_bottom_goal:
        r1, r2, done = -1.0, +1.0, True

    return self.get_state(), r1, r2, done
```

encode_for_j1() y encode_for_j2(): Codifican el estado para cada jugador

python

```
def encode_for_j1(self, s):
    ar1, ac1, ar2, ac2, pr, pc, pd = s
    dr = np.clip(pr - ar1, -2, 2)
    dc = np.clip(pc - ac1, -2, 2)
    return (int(dr), int(dc), int(pd))
```

4.3 Clase Agent

Implementa el algoritmo Q-Learning para cada jugador.

Atributos

python

```
class Agent:
    def __init__(self, name, alpha=0.2, gamma=0.95,
                 eps0=0.9, eps_min=0.05, eps_decay=0.995):
        self.name = name
        self.Q = {}          # Tabla Q (diccionario)
        self.alpha = alpha   # Tasa de aprendizaje
        self.gamma = gamma   # Factor de descuento
```

```
    self.eps = eps0      # Epsilon inicial
    self.eps_min = eps_min
    self.eps_decay = eps_decay
```

Métodos Principales

q_value(s, a): Obtiene el valor Q de un par estado-acción

python

```
def q_value(self, s, a):
    return self.Q.get((s, a), 0.0)  # 0.0 si no existe
```

best_action(s): Encuentra la mejor acción para un estado

python

```
def best_action(self, s):
    # Calcular Q para todas las acciones
    qs = np.array([self.q_value(s, a) for a in ACTIONS])

    # Encontrar el máximo
    m = np.max(qs)

    # Si hay empate, elegir aleatoriamente
    best_as = np.where(qs == m)[0]
    a = int(np.random.choice(best_as))

    return a, float(m)
```

choose_action(s): Política ε-greedy

python

```
def choose_action(self, s):
    if random.random() < self.eps:
        return random.choice(ACTIONS)  # Exploración
    return self.best_action(s)[0]      # Explotación
```

update(s, a, r, s_next): Actualización Q-Learning

python

```python
def update(self, s, a, r, s_next):
    # Mejor valor del siguiente estado
    _, max_next = self.best_action(s_next)

    # Valor actual
    old = self.q_value(s, a)

    # TD Target
    td_target = r + self.gamma * max_next

    # Actualización
    new = old + self.alpha * (td_target - old)

    # Guardar
    self.Q[(s, a)] = new
```

decay_epsilon(): Reduce epsilon después de cada episodio

python

```python
def decay_epsilon(self):
    self.eps = max(self.eps_min, self.eps * self.eps_decay)
```

Explicación de la Actualización Q-Learning

La actualización se descompone en:

1. **old = Q(s, a):** Estimación actual del valor
2. **td_target = r + γ max Q(s', a'):** Objetivo basado en la recompensa real
3. **td_error = td_target - old:** Diferencia temporal
4. **new = old + α × td_error:** Nueva estimación

Esto hace que Q converja hacia el valor óptimo con el tiempo.

4.4 Clase Renderer

Maneja la visualización con Pygame.

Inicialización

python

```
class Renderer:
    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode((W, H))
        pygame.display.set_caption("Hockey 2D – POO")
```

Conversión de Coordenadas

python

```
def to_xy(self, r, c):
    x = MARGIN + c * CELL + CELL // 2
    y = MARGIN + r * CELL + CELL // 2
    return x, y
```

Dibujado del Grid

python

```
def draw_grid(self):
    # Fondo
    self.screen.fill((25, 28, 35))

    # Líneas horizontales
    for r in range(N_ROWS+1):
        pygame.draw.line(screen, (70,70,80),
                         (MARGIN, MARGIN+r*CELL),
                         (MARGIN+N_COLS*CELL, MARGIN+r*CELL), 1)

    # Líneas verticales
    for c in range(N_COLS+1):
        pygame.draw.line(screen, (70,70,80),
                         (MARGIN+c*CELL, MARGIN),
                         (MARGIN+c*CELL, MARGIN+N_ROWS*CELL), 1)

    # Dibujar arcos (rectángulos de color)
    # ... código de arcos ...
```

Dibujado de Elementos

python

```python
def draw(self, env, info=""):
    self.draw_grid()

    # Jugador 1 (azul)
    pygame.draw.circle(screen, (70,170,255),
                       self.to_xy(env.ar1, env.ac1), CELL//3)

    # Jugador 2 (naranja)
    pygame.draw.circle(screen, (255,170,70),
                       self.to_xy(env.ar2, env.ac2), CELL//3)

    # Puck (blanco)
    pygame.draw.circle(screen, (240,240,240),
                       self.to_xy(env.pr, env.pc), CELL//6)

    # Texto informativo
    font = pygame.font.SysFont("arial", 18)
    screen.blit(font.render(info, True, (220,220,230)),
                (MARGIN, H - int(MARGIN*0.6)))

    pygame.display.flip()
```

---

5. Proceso de Entrenamiento

5.1 Función de Entrenamiento

python

```python
def run_training(episodes=2000, max_steps=180):
    # Inicialización
    env = HockeyEnv()
    render = Renderer()
    j1 = Agent("J1")
    j2 = Agent("J2")
```

```
rewards1, rewards2 = [], []

# Bucle de episodios
for ep in range(1, episodes+1):
    s = env.reset()
    R1 = R2 = 0.0
    render_now = (ep % RENDER_EVERY == 0)

    # Bucle de pasos
    for t in range(max_steps):
        # Codificar estado
        s1 = env.encode_for_j1(s)
        s2 = env.encode_for_j2(s)

        # Elegir acciones
        a1 = j1.choose_action(s1)
        a2 = j2.choose_action(s2)

        # Ejecutar paso
        s_next, r1, r2, done = env.step(a1, a2)

        # Acumular recompensas
        R1 += r1
        R2 += r2

        # Actualizar Q
        j1.update(s1, a1, r1, env.encode_for_j1(s_next))
        j2.update(s2, a2, r2, env.encode_for_j2(s_next))

        # Renderizar si corresponde
        if render_now:
            render.draw(env, f"TRAIN EP={ep} step={t}")

        s = s_next
        if done: break
```

```
        # Decaimiento de epsilon
        j1.decay_epsilon()
        j2.decay_epsilon()

        # Guardar recompensas
        rewards1.append(R1)
        rewards2.append(R2)

    return env, render, j1, j2, rewards1, rewards2
```

5.2 Parámetros de Entrenamiento

Parámetro	Valor	Descripción
Episodios	2000	Número de partidas completas
Max Steps	180	Pasos máximos por episodio
Alpha (α)	0.2	Tasa de aprendizaje
Gamma (γ)	0.95	Factor de descuento
Epsilon inicial	0.9	Exploración inicial (90%)
Epsilon mínimo	0.05	Exploración mínima (5%)
Decaimiento ε	0.995	Factor de reducción de epsilon

5.3 Evolución del Entrenamiento

Fase Inicial (Episodios 1-500)

- **Epsilon alto (≈0.9):** Mucha exploración
- **Comportamiento:** Movimientos casi aleatorios
- **Tabla Q:** Mayormente vacía, valores cercanos a 0
- **Recompensas:** Altamente variables y negativas

Fase Media (Episodios 500-1500)

- **Epsilon medio (≈0.3-0.6):** Balance exploración/explotación
- **Comportamiento:** Emergencia de patrones básicos
- **Tabla Q:** Se puebla con valores significativos
- **Recompensas:** Menor variabilidad, algunos episodios positivos

Fase Final (Episodios 1500-2000)

- **Epsilon bajo (≈0.05-0.2):** Mayormente explotación
- **Comportamiento:** Estrategias consolidadas
- **Tabla Q:** Valores estabilizados
- **Recompensas:** Convergencia visible

5.4 Estrategias Emergentes

A medida que avanzan los episodios, los agentes desarrollan comportamientos no programados explícitamente:

1. **Persecución del Puck:** Moverse hacia la posición del puck
2. **Interceptación:** Anticipar la trayectoria del puck
3. **Defensa de Arco:** Posicionarse cerca del arco propio cuando el puck se acerca
4. **Contra-ataque:** Golpear el puck hacia el arco contrario

Estos comportamientos emergen naturalmente del proceso de aprendizaje guiado por las recompensas.

---

6. Resultados

6.1 Gráfica de Recompensas

La gráfica generada muestra las recompensas acumuladas por episodio para ambos jugadores:

python

```
plt.plot(rewards1, label="Jugador 1")
plt.plot(rewards2, label="Jugador 2")
plt.legend()
plt.grid(True)
plt.title("Recompensas por episodio")
plt.xlabel("Episodio")
plt.ylabel("Recompensa Acumulada")
plt.show()
```

6.2 Observaciones Clave

Predominio de Recompensas Negativas

Las recompensas son mayormente negativas debido a:

- **Penalización por paso:** -0.1 en cada turno
- **Episodios largos:** Hasta 180 pasos posibles
- **Goles escasos:** Especialmente al inicio del entrenamiento

Cálculo típico:

- Episodio de 100 pasos sin goles: -0.1 × 100 = -10.0
- Episodio de 80 pasos con 1 gol: -0.1 × 80 + 1.0 = -7.0

Reducción de Variabilidad

Con el tiempo, la varianza de las recompensas disminuye, indicando:

- Políticas más estables
- Menor aleatoriedad en las acciones
- Convergencia del aprendizaje

Simetría entre Agentes

Ambos jugadores muestran curvas similares, lo cual es esperado dado que:

- Tienen los mismos hiperparámetros
- Enfrentan el mismo problema (simétrico)
- Aprenden simultáneamente

6.3 Visualización de Partidas

La función `run_demo` permite observar el comportamiento aprendido:

python

```
def run_demo(env, render, j1, j2, episodes=5, max_steps=180):
    for demo in range(episodes):
        s = env.reset()

        for t in range(max_steps):
            s1 = env.encode_for_j1(s)
            s2 = env.encode_for_j2(s)

            # Usar política greedy (sin exploración)
            a1, v1 = j1.best_action(s1)
            a2, v2 = j2.best_action(s2)
```

```
        s_next, r1, r2, done = env.step(a1, a2)
        render.draw(env, f"DEMO EP={demo+1} | V1={v1:.2f} V2={v2:.2f}")

        pygame.time.delay(RENDER_DELAY_DEMO)
        s = s_next
        if done:
            pygame.time.delay(500)
            break
```

En el modo demo:

- **Sin exploración:** Solo se usa la mejor acción conocida
- **Pausa visible:** 150ms entre frames para observación
- **Valores Q mostrados:** Confianza del agente en sus decisiones

---

7. Conclusiones

7.1 Logros del Proyecto

1. **Implementación Exitosa de Q-Learning Multi-Agente**
    - Dos agentes aprenden simultáneamente en un entorno competitivo
    - Convergencia observable del proceso de aprendizaje
    - Desarrollo de estrategias emergentes sin programación explícita
2. **Diseño de Entorno Efectivo**
    - Sistema de estados codificados reduce complejidad computacional
    - Balance adecuado de recompensas guía el aprendizaje
    - Dinámica del puck añade imprevisibilidad realista
3. **Visualización Educativa**
    - Interfaz gráfica clara con Pygame
    - Observación del proceso de entrenamiento en tiempo real
    - Modo demo para evaluar políticas aprendidas
4. **Código Modular y Extensible**
    - Estructura POO facilita modificaciones
    - Separación clara de responsabilidades
    - Documentación interna mediante comentarios

7.2 Análisis de Convergencia

¿Los Agentes Aprenden?

Evidencia de Aprendizaje:

- ✓ Reducción de variabilidad en recompensas
- ✓ Aumento de valores Q promedio
- ✓ Comportamientos no aleatorios en modo demo
- ✓ Tabla Q se puebla con valores no-cero

Limitaciones de Convergencia:

- La convergencia no está garantizada en multi-agente
- Puede alcanzar equilibrios subóptimos
- Sensible a inicialización y hiperparámetros

Tiempo de Convergencia

Con los parámetros actuales:

- **Aprendizaje básico:** ~500 episodios
- **Estabilización:** ~1500 episodios
- **Refinamiento:** 1500-2000 episodios

Factores que afectan:

- Tasa de aprendizaje α
- Factor de descuento γ
- Estrategia de decaimiento de ε

Conclusión Final

Este proyecto demuestra exitosamente la aplicación de **Aprendizaje por Refuerzo Multi-Agente** en un entorno competitivo. A través de Q-Learning Independiente, dos agentes aprenden estrategias de juego sin programación explícita de reglas.

Los resultados muestran que:

- Los agentes desarrollan comportamientos emergentes
- El aprendizaje converge en aproximadamente 1500-2000 episodios
- Las políticas aprendidas son observables y evaluables

Este trabajo sirve como base educativa sólida para comprender RL

Visualización del Juego: