Universidad Mayor de San Andrés

Facultad de Ciencias Puras y Naturales
Carrera de Informática

Proyecto Final:

Ratón vs Zorro con Aprendizaje por Refuerzo

Autor: Ian Ezequiel Salinas Condori

Gestión 2025

Contents

1. Introducción

El presente proyecto desarrolla un entorno interactivo basado en Pygame donde un jugador controla a un ratón dentro de un laberinto, mientras un zorro controlado por inteligencia artificial intenta atraparlo. Este proyecto combina conceptos teóricos y prácticos de Aprendizaje por Refuerzo (Reinforcement Learning, RL), Procesos de Decisión de Markov (MDP) y modelado de comportamientos adaptativos mediante un sistema adicional llamado *Director IA*.

El objetivo principal es demostrar cómo un agente inteligente puede aprender una política eficiente de persecución empleando técnicas como SARSA y Q-Learning, ampliadas mediante un mecanismo de toma de decisiones de nivel superior inspirado en el sistema de inteligencia artificial de *Alien: Isolation*.

Para evaluar el progreso del agente, se implementaron métricas avanzadas y gráficas que permiten analizar el comportamiento del sistema en el tiempo.

2. Objetivo del Proyecto

- Diseñar un entorno interactivo controlado mediante Aprendizaje por Refuerzo.
- Implementar dos algoritmos RL: SARSA (on-policy) y Q-Learning (off-policy).
- Desarrollar un Director IA capaz de ajustar dinámicamente el comportamiento del agente.
- Medir el desempeño del agente mediante gráficos, métricas y registros de tiempo.
- Analizar el proceso de aprendizaje y la calidad de las decisiones del zorro a lo largo del entrenamiento.

3. Explicación Detallada de los Códigos del Proyecto

En esta sección se describe en profundidad el funcionamiento interno de cada módulo del proyecto. El objetivo es que el lector pueda entender no sólo "qué hace" el código, sino también "por qué" se diseñó de esa manera y cómo se relaciona con los conceptos teóricos de Aprendizaje por Refuerzo y Procesos de Decisión de Markov (MDP).

3.1. 1. Archivo `config.py`: Parámetros Globales y Diseño del Experimento

El archivo `config.py` concentra todas las constantes y parámetros de configuración que controlan tanto la parte gráfica del juego como el comportamiento del algoritmo de Aprendizaje por Refuerzo.

1.1. Parámetros del entorno gráfico

- `TAMANO_CELDA`: define el tamaño en píxeles de cada celda del laberinto. En el proyecto se usa 32 píxeles, lo cual determina el tamaño de los sprites del ratón, zorro y queso.
- `FILAS` y `COLUMNAS`: fijan el tamaño de la matriz del laberinto (por ejemplo 15×15).
- `ANCHO_PANTALLA` y `ALTO_PANTALLA`: se calculan a partir de las celdas e incluyen un área superior adicional para el HUD (tiempo y valor de ϵ).

1.2. Parámetros de color y estados del juego

Se definen colores en formato RGB (por ejemplo `AZUL`, `ROJO`, etc.) para mantener el estilo visual homogéneo, y constantes simbólicas como `MENU_PRINCIPAL`, `JUGANDO`, `VICTORIA_RATON` y `VICTORIA_ZORRO`. Estas constantes permiten implementar una máquina de estados sencilla dentro del bucle principal.

1.3. Parámetros de Aprendizaje por Refuerzo

- `ALPHA`: tasa de aprendizaje α, cuánto se ajustan los valores de $Q(s, a)$ en cada actualización.
- `GAMMA`: factor de descuento γ, que pondera la importancia de las recompensas futuras.
- `EPSILON_INICIAL`, `EPSILON_MIN` y `EPSILON_DECAY`: controlan la política ϵ-greedy; el agente explora más al inicio y cada vez explora menos.
- `ALGORITMO_RL`: permite elegir si el zorro usa **SARSA** o **Q-Learning** sin modificar el resto del código.

1.4. Archivos de salida y frecuencia de movimiento

- Rutas como `RUTA_APRENDIZAJE` y `RUTA_TIEMPOS` controlan dónde se guardarán los datos de entrenamiento para generar luego las gráficas.
- `INTERVALO_MOVIMIENTO_ZORRO` fija cada cuántos frames se mueve el zorro. Valores pequeños lo hacen más agresivo (se mueve más seguido).

En resumen, `config.py` actúa como una "capa de diseño experimental" del proyecto.

3.2. 2. Archivo `laberinto.py`: Representación del Mundo y Validación Espacial

El archivo `laberinto.py` modela el entorno físico donde ocurre el proceso de decisión de Markov. Aquí se definen tres laberintos fijos, uno por cada nivel del juego.

2.1. Representación del laberinto

Cada nivel es una lista de listas de enteros:

$$\text{LABERINTO\_NIVEL\_k}[i][j] = \begin{cases} 0 & \text{si la celda es transitable (camino),} \\ 1 & \text{si la celda es un muro (pared).} \end{cases}$$

Esta discretización convierte el entorno continuo en un espacio de estados finito, adecuado para un MDP.

2.2. Selección de nivel

La función:

```
def set_nivel(nivel: int):
global LABERINTO, NIVEL_ACTUAL
NIVEL_ACTUAL = nivel
LABERINTO = LABERINTOS[nivel - 1]
```

actualiza el laberinto global según la elección del jugador en el menú. De esta manera, los mismos algoritmos de RL funcionan sin cambios sobre distintos mapas.

2.3. Función `es_celda_valida`

```
def es_celda_valida(x, y):
if x < 0 or x >= len(LABERINTO): return False
if y < 0 or y >= len(LABERINTO[0]): return False
return LABERINTO[x][y] == 0
```

Esta función garantiza que el agente respete las fronteras y los muros del entorno. Desde la perspectiva de RL, determina el conjunto de acciones válidas desde un estado dado.

2.4. Función `celda_libre_mas_cercana`

Cuando se desea colocar al ratón, al zorro o al queso en una posición inicial aproximada, esta función realiza una búsqueda BFS alrededor de una coordenada objetivo hasta encontrar una celda con valor 0 (camino). Esto evita inicializaciones inválidas sobre muros y garantiza que el entorno esté siempre jugable.

2.5. Función `dibujar_laberinto`

Se recorre la matriz y se dibuja un rectángulo para cada celda, usando negro para muros y blanco para caminos. Esta separación clara entre representación lógica (matriz de 0/1) y representación gráfica (rectángulos en Pygame) hace el código más limpio y modificable.

3.3. 3. Archivo `director_ia.py`: Sistema de Director IA y Modos de Comportamiento

El **Director IA** es una capa de inteligencia superior que no reemplaza al agente RL, sino que lo guía. Esta idea está inspirada en juegos como *Left 4 Dead* o *Alien: Isolation*, donde un "director" ajusta la dificultad en tiempo real.

3.1. Variables internas del Director

- `tension`: valor continuo entre 0 y 1 que mide lo "peligroso" que está el entorno. Aumenta cuando el zorro ve al ratón y disminuye cuando lo pierde.
- `ultima_pos_raton_vista`: almacena la última celda donde el director tuvo línea de visión directa con el ratón.

- `historial_raton`: cola (`deque`) de posiciones recientes del ratón, usada para construir un **heatmap** de zonas calientes.
- `zonas_calientes`: diccionario que cuenta cuántas veces el ratón ha pasado por cada celda.

3.2. Línea de visión y distancia de Manhattan

La función `linea_de_vision(a,b)` determina si hay muros entre dos celdas en la misma fila o columna. La distancia de Manhattan:

$$d(a, b) = |a_x - b_x| + |a_y - b_y|$$

se utiliza tanto para clasificar la cercanía como para definir modos como CAZANDO o ACECHANDO.

3.3. Modos de comportamiento

En `actualizar_estado`, el Director IA decide un modo y un objetivo:

- **CAZANDO**: si el zorro está muy cerca del ratón (distancia ≤ 1), el objetivo es la posición actual del ratón.
- **ACECHANDO**: el zorro ve al ratón a media distancia; el director mantiene una alta tensión y incentiva una persecución directa.
- **RASTRO_RECIENTE**: si se perdió de vista hace poco, el zorro se dirige a la última posición conocida.
- **ZONA_CALIENTE**: si la tensión es alta pero no se ve al ratón, el director envía al zorro hacia zonas donde el ratón suele pasar.
- **EXPLORANDO**: comportamiento base cuando no hay información especial; el objetivo vuelve a ser simplemente la posición actual del ratón.

El valor de ϵ efectivo que usa el zorro puede modificarse según el modo (por ejemplo, menos exploración cuando está CAZANDO).

3.4. 4. Archivo `rl_agent.py`: Implementación de SARSA y Q-Learning

Este archivo constituye el corazón algorítmico del proyecto. Representa la función de valor $Q(s, a)$ mediante una tabla que se aproxima con un diccionario de Python.

4.1. Representación de la Q-table

```
self.q_table = {}  # diccionario: estado -> [Q_up,
    Q_down, Q_left, Q_right]
```

Cada estado es una tupla con información relativa (dx, dy) y presencia de muros alrededor. Si un estado no se ha visitado antes, se inicializa con cuatro ceros.

4.2. Política ϵ-greedy

```
def seleccionar_accion_con(self, epsilon, estado):
if random.uniform(0, 1) < epsilon:
return accion_aleatoria
else:
return accion_con_mejor_Q
```

Esto implementa la exploración de nuevas acciones (cuando cae en el ramo aleatorio) y la explotación del conocimiento actual (cuando elige la acción de mayor Q).

4.3. Paso de entrenamiento

La función clave es `paso_entrenamiento`, que realiza:

1. Obtiene el estado actual.
2. Elige una acción con la política ϵ-greedy ajustada por el modo del Director.
3. Calcula la nueva posición del zorro.
4. Asigna una recompensa:
    - Penalización fuerte si choca contra una pared.
    - Penalización pequeña por cada paso.
    - **Recompensa de shaping** si se acerca al objetivo definido por el Director IA.

- Gran recompensa si logra atrapar al ratón.

5. Calcula el nuevo estado.
6. Aplica la actualización de Bellman:
   - Si se usa SARSA, se calcula también la siguiente acción a' y se usa $Q(s', a')$.
   - Si se usa Q-Learning, se usa el máximo $\max_{a'} Q(s', a')$.
7. Actualiza el valor de ϵ reduciéndolo progresivamente.
8. Registra la recompensa en un archivo CSV para futuros gráficos.

De esta forma se implementa un MDP donde cada paso de juego corresponde a una transición (s, a, r, s').

3.5. 5. Archivo `game.py`: Bucle Principal y Lógica de Juego

El archivo `game.py` es el punto de entrada funcional del sistema. Orquesta la interacción entre:

- Pygame (renderizado y eventos de teclado),
- el entorno (laberinto),
- el agente RL (zorro),
- el Director IA,
- y el jugador (ratón).

5.1. Menú principal y selección de nivel

En el estado `MENU_PRINCIPAL`, el juego:

- Dibuja un panel con opciones de nivel (1, 2 o 3).
- Permite elegir ver las gráficas de aprendizaje.
- Permite salir del programa.

Al presionar una tecla de nivel se llama a `inicializar_nivel(nivel_actual)`, que:

- Llama a `set_nivel` para cambiar de laberinto.
- Calcula posiciones válidas para ratón, zorro y queso.
- Reinicia el agente RL y el Director IA.
- Inicializa el contador de tiempo de la partida.

5.2. Bucle de juego (estado `JUGANDO`)

En cada iteración del bucle:

1. Se leen los eventos de teclado para mover al ratón.
2. Cada cierto número de frames, el Director IA calcula un modo y un objetivo.
3. El agente RL realiza un `paso_entrenamiento` usando esa información.
4. Se verifica si el zorro atrapó al ratón (victoria del zorro) o si el ratón llegó al queso (victoria del ratón).
5. Se actualiza el HUD con el tiempo y el valor actual de ϵ.
6. Se redibuja el laberinto, el ratón, el zorro y el queso.

Cuando ocurre una victoria, se pasa a un estado de `VICTORIA_RATON` o `VICTORIA_ZORRO`, donde se muestran mensajes y opciones para reiniciar nivel, volver al menú o salir.

5.3. Registro de tiempos

Cada partida se registra en un archivo de texto con el formato:

$$\text{GANADOR}, \text{TIEMPO}$$

Lo cual sirve para analizar posteriormente el tiempo medio de captura o escape.

3.6. 6. Archivo `analisis/graficas.py`: Métricas y Evaluación del Aprendizaje

Finalmente, el módulo `analisis/graficas.py` toma los archivos de registro generados por `rl_agent.py` y `game.py` y construye varias métricas visuales.

6.1. Recompensa media móvil

Se carga el archivo `aprendizaje_rl.csv` con columnas `step` y `reward`. Para suavizar el ruido se calcula una media móvil:

$$\overline{r}_t = \frac{1}{N} \sum_{k=t-N+1}^{t} r_k,$$

para una ventana N (por ejemplo, $N = 200$). Esta serie suavizada muestra la tendencia general del desempeño del agente a lo largo de los pasos.

6.2. Recompensa vs castigo acumulado

Se separan las recompensas positivas y negativas:

$$r_t^+ = \max(r_t, 0), \qquad r_t^- = \min(r_t, 0),$$

y luego se calculan las sumas acumuladas:

$$R_t^+ = \sum_{k=1}^{t} r_k^+, \qquad R_t^- = \sum_{k=1}^{t} r_k^-.$$

Gráficamente, estas curvas permiten entender el balance entre buenas y malas decisiones.

6.3. Win-rate y tiempos promedio

A partir de `registroTiempo.txt`, se calcula:

- El win-rate del zorro en ventanas de partidas.
- El tiempo promedio cuando gana el zorro.
- El tiempo promedio cuando gana el ratón.

Estos valores son cruciales para interpretar si el agente "se vuelve más peligroso" y si atrapa cada vez más rápido al jugador.

En conjunto, todos estos módulos forman una arquitectura coherente donde:

1. `laberinto.py` define el entorno MDP.
2. `rl_agent.py` aprende la política mediante Bellman.

3. `director_ia.py` actúa como un controlador de alto nivel.

4. `game.py` conecta al jugador humano con el agente.

5. `analisis/graficas.py` mide y visualiza el aprendizaje.

4. Análisis de Resultados y Gráficas

4.1. Recompensa Media Móvil

La figura muestra cómo el zorro reduce errores con el tiempo. La recompensa media inicial es muy negativa debido a exploración, pero converge a valores moderadamente negativos que reflejan una política estable.

4.2. Recompensa vs Castigos

El castigo acumulado es alto al comienzo debido a colisiones y exploración. Con el tiempo, la pendiente disminuye, indicando:

- Menos decisiones equivocadas,
- Mejor entendimiento del entorno,
- Políticas más orientadas al objetivo.

Esto demuestra aprendizaje real.

5. Conclusión

El sistema demuestra que:

- El zorro aprende estrategias reales de persecución.
- El Director IA potencia la inteligencia del agente.
- Las gráficas confirman una mejora continua durante el entrenamiento.
- La arquitectura modular permite ampliar niveles, comportamientos y estrategias.

El proyecto cumple los objetivos académicos y muestra el poder del Aprendizaje por Refuerzo en entornos interactivos.

Anexos

A. Código del Agente RL

```
class AgenteRL:
def paso_entrenamiento(self, pos_zorro, pos_raton,
   modo, objetivo):
estado = self.obtener_estado(pos_zorro, pos_raton)
accion = self.seleccionar_accion_con(self.epsilon,
   estado)
...
```

B. Director IA

```
if dist <= 1:
modo = "CAZANDO"
elif ve_raton and dist < 6:
modo = "ACECHANDO"
elif self.ultima_pos and self.tiempo_sin_ver < 10:
modo = "RASTRO_RECIENTE"
...
```

C. Movimiento del Zorro en game.py

```
modo, objetivo = director.actualizar_estado(
   pos_zorro, pos_raton)
pos_zorro, atrapado = agente.paso_entrenamiento(
pos_zorro, pos_raton, modo, objetivo
)
```