# Machine Learning

MEMORIA
PRÁCTICA 2

Javier Álvarez Pérez
Ana Ordóñez Gragera
Fabio Elías Rengifo García

**DESARROLLO DE JUEGOS CON INTELIGENCIA ARTIFICIAL**
2024

# Índice

# Introducción

El objetivo principal del proyecto es implementar, entrenar y evaluar un agente basado en el algoritmo de aprendizaje por refuerzo **Q-Learning** en un entorno dinámico y no determinista de **Unity3D**, haciendo uso del lenguaje de programación **C#**.

Específicamente, se busca que el agente desarrolle un comportamiento inteligente que le permita:

- Tomar decisiones óptimas en cada estado.
- Minimizar el riesgo de ser alcanzado por el enemigo.
- Maximizar el número de pasos antes de alcanzar un estado terminal.

Para conseguir cumplir los objetivos propuestos, se debe realizar la implementación de los siguientes scripts: **State**, clase que representa un estado del agente en el entorno; **QTable**, que almacena y gestiona los valores Q asociados a cada estado y **QMindTrainer**, que controla el proceso de aprendizaje.

# *CLASES:*

## *State*

La clase State nos permite tener una representación del estado del agente en el entorno. Su principal función es generar un identificador único (**idState**) que permita distinguir un estado concreto durante el aprendizaje. Este identificador se emplea como clave en la clase **QTable**, permitiendo registrar y consultar los valores Q asociados a las distintas acciones posibles en ese estado.

Para conseguir hacer su correcta implementación, se han planteado los siguientes atributos de la clase:

- **agentPosition**: objeto de tipo **CellInfo** que representa la posición actual del agente en el entorno.
- **otherPosition**: objeto de tipo **CellInfo** que representa la posición actual del enemigo en el entorno.
- **distance**: float que calcula la distancia de **Manhattan** entre el agente y el enemigo.
- **enemyNear**: booleano que indica si el enemigo está **cerca del agente**.
- **up**, **down**, **left**, **rigth**: objetos de tipo **CellInfo**. Representan las celdas adyacentes al agente en las direcciones norte, sur, oeste y este, respectivamente.
- **nWall**, **sWall**, **eWall**, **wWall**: booleanos que indican si hay un muro al norte, sur, este o oeste del agente, respectivamente.
- **nEnemy**: booleano que indica si el enemigo está al norte del agente.
- **eEnemy**: booleano que indica si el enemigo está al este del agente.
- **idState**: string que se usa como identificador único generado para el estado actual.

El constructor principal de la clase se forma con la posición de nuestro agente, agentPosition; la posición del enemigo, otherPosition y, además, la información del mundo.

```
public State(CellInfo agentPosition,
    CellInfo otherPosition, WorldInfo worldInfo)
```

Dentro de este constructor, se hace lo siguiente:

- Se calcula la distancia entre el agente y el enemigo utilizando la distancia de Manhattan.
- También, identifica si hay muros alrededor del agente en las direcciones norte, sur, este y oeste, verificando si las celdas adyacentes son transitables.
- Determina la dirección relativa del enemigo con respecto al agente, estableciendo si se encuentra al norte o al sur, y al este o al oeste.
- Una vez recopilada toda esta información, genera un identificador único, **idState,** que representa el estado actual. Esto se hace a través del método **GenerateId().**

```
private string GenerateId()
{
    return $"{(nWall ? 1 : 0)}" +
           $"{(sWall ? 1 : 0)}" +
           $"{(eWall ? 1 : 0)}" +
           $"{(wWall ? 1 : 0)}" +
           $"{(nEnemy ? 1 : 0)}" +
           $"{(eEnemy ? 1 : 0)}" +
           $"{(enemyNear ? 1 : 0)}";
}
```

# CLASES:

# QTable

La clase QTable gestiona la tabla Q empleada en el aprendizaje por refuerzo, permitiendo almacenar, actualizar y recuperar los valores Q asociados a cada estado. Estos se identifican mediante un ID único, y sus valores Q se representan como un array de floats, donde cada posición corresponde a una acción. La clase ofrece métodos para guardar y cargar la tabla Q en un archivo CSV.

Se han añadido funcionalidades que nos facilitarán trabajar con el entorno, como funciones que devuelven la acción con el mayor valor Q, que obtienen el valor Q de una acción específica o que actualizan los valores Q cuando el agente aprende. Si un estado no existe en la tabla, se inicializa automáticamente con valores Q en cero.

Los atributos de la clase son los siguientes:

- **actions**: integer que representa el número de acciones posibles.
- **qTable**: atributo de tipo **Dictionary<string, float[]>** que representa la tabla Q donde cada clave es un **idState** y el valor es un array de floats que guarda los valores Q correspondientes a cada acción en ese estado.

El constructor principal de la clase no necesita información de entrada, ya que solamente inicializa los atributos.

```
public QTable()
{
    actions = 4;
    qTable= new Dictionary<string, float[]>();
}
```

A continuación, se verán los distintos métodos de la clase y sus funcionalidades:

- **Save()**. Guarda la tabla Q en un archivo CSV mediante el método **ConvertCsv()**.

```
public void Save()
{
    string filePath = @"Assets/Scripts/GrupoB/TablaQ.csv";

    // Escribir el archivo
    File.WriteAllLines(filePath, ConvertCsv(qTable));
}
```

- **ConvertCsv()**. Convierte el contenido de la tabla Q en una lista de líneas en formato CSV. Cada línea contiene el ID del estado y sus valores Q separados por comas.
- **Load()**. Carga la tabla Q desde un archivo CSV. Lee el archivo línea por línea, separa el ID del estado y sus valores Q, y los almacena en el diccionario qTable.
- **GetAction()**. Devuelve la mejor acción para un estado dado, es decir, el de mayor valor Q. Si el estado no existe en la tabla, se inicializa con valores Q en cero. Recorre las acciones asociadas al estado y selecciona la de mayor valor.
- **GetQValue()**. Devuelve el valor Q de una acción específica en un estado dado. Si el estado no está registrado, se inicializa con valores Q en cero.
- **GetMaxQValue()**. Devuelve el valor Q máximo de todas las acciones disponibles en un estado dado. Si el estado no existe en la tabla, se inicializa con valores Q en cero.
- **UpdateQValue()**. Actualiza el valor Q de una acción específica en un estado dado. Si el estado no está registrado, se inicializa con valores Q en cero.

# *CLASES:*
# *QMindTrainer*

El objetivo de la clase es entrenar al agente para que aprenda a moverse en un entorno y huir del enemigo según las actualizaciones de la tabla Q, que almacena los valores de cada acción posible en diferentes estados. La clase gestiona el entorno de entrenamiento, selecciona acciones, calcula recompensas basadas en la proximidad al enemigo, y actualiza los valores de la tabla Q mediante la regla de aprendizaje.

A continuación, se observan los atributos de la clase adicionales:

- **_qTable**. Objeto de tipo QTable, almacena los valores Q para cada estado y acción.
- **_episodeCount**. Atributo de tipo entero que lleva el conteo del total de episodios realizados durante el entrenamiento.
- **_stepCount**. Entero que cuenta el número total de pasos realizados desde el inicio del entrenamiento.
- **totalReward**. Atributo de tipo float que almacena la recompensa total acumulada durante el entrenamiento.
- **terminal_state**. Booleano que indica si el agente ha alcanzado un estado terminal.
- **epsilon**. Atributo de tipo float que representa la tasa de exploración.
- **alpha**. Atributo de tipo float que representa la tasa de aprendizaje.
- **gamma**. Atributo de tipo float que es el factor de descuento que pondera el valor de las recompensas futuras en el cálculo del valor Q.

La implementación clase QMindTrainer se ha conseguido a través de diferentes métodos:

- **Initialize()**. Inicializa el entorno de entrenamiento, carga la tabla Q, establece las posiciones iniciales del agente y el enemigo, y configura los parámetros.
- **DoStep()**. Realiza un paso de entrenamiento o simulación. Si se ha alcanzado un estado terminal, reinicia el entorno.

```
if (terminal_state)
{
    ReturnAveraged = (float)(ReturnAveraged * 0.9 + Return * 0.1);
    OnEpisodeFinished?.Invoke(this, EventArgs.Empty);
    ResetEnvironment();
}
```

El método **ResetEnviroment()** nos ayuda a reiniciar el entorno cuando se alcanza un estado terminal, a actualizar los contadores y guardar la tabla Q cada cierto número de episodios. Si no entra en un estado terminal, calcula el estado actual, selecciona una acción y actualiza el entorno:

```
State state = new State(AgentPosition, OtherPosition, _worldInfo);
int action = selectAction(state);
(CellInfo newAgentPosition, CellInfo newOtherPosition) = UpdateEnvironment(action);
```

Con el método **selectAction()** se aplica la política epsilon-greedy para seleccionar la acción. **UpdateEnviroment()** calcula nuevas posiciones del agente y el enemigo tras aplicar la acción. **DoStep()** continua calculando la recompensa, y cambia la tabla Q usando la regla de aprendizaje.

```
float reward = CalculateReward(newAgentPosition, newOtherPosition);
totalReward += reward;
Return = Mathf.Round(totalReward * 10) / 10;
UpdateQtable(state, action, reward, nextState);
```

**CalculateReward()** calcula la recompensa en función de la nueva posición del agente. Esta penaliza fuertemente si el agente se mueve en una celda no caminable o es capturado

por el enemigo. Recompensa si el agente se aleja del enemigo y penaliza si se acerca.

El método **UpdateQTable() a**ctualiza el valor Q de una acción específica en un estado usando la ecuación del Q-Learning:

$$Q'(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{\forall a'} Q(s',a'))$$

Por último, se actualizan las posiciones del agente y del enemigo a las nuevas posiciones. Además, se incrementa el contador de pasos.

```
AgentPosition = newAgentPosition;
OtherPosition = newOtherPosition;

_stepCount++;
CurrentStep = _stepCount;
```

# CLASES:

# QMindTester

La clase se encarga de usar una tabla de Q preexistente para determinar el siguiente movimiento del agente.

Los atributos de la clase son:

- **_worldInfo.** Es un elemento de tipo WorldInfo que contiene información sobre el entorno en el que se encuentra el agente.
- **_qTable.** Objeto de tipo QTable, almacena y maneja la tabla de Q.
- **currentState.** Es un objeto de tipo State que representa el estado actual del agente.

Esta clase cuenta con dos métodos:

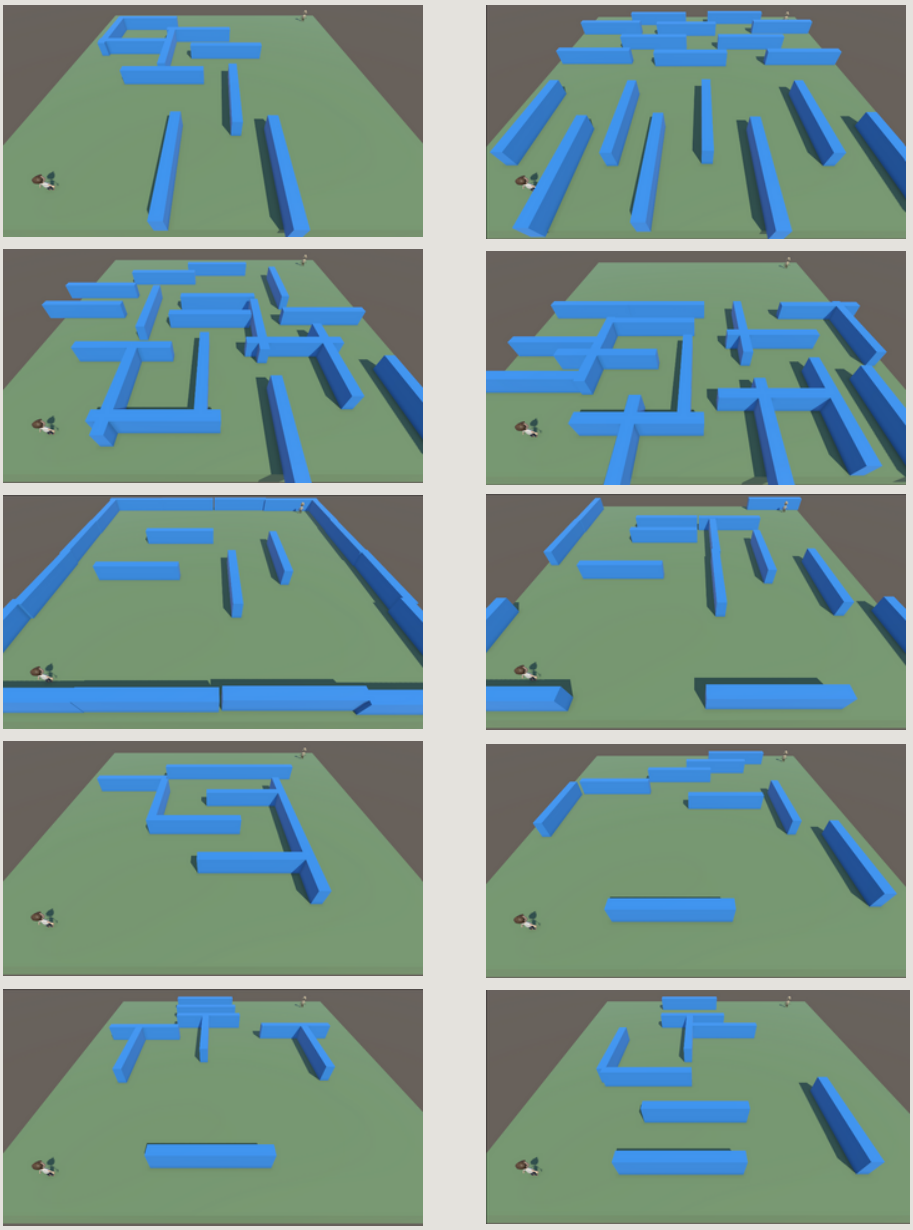
- **Initialize().** Carga la tabla de Q desde un archivo.
- **GetNextStep()**. Recibe las posiciones actuales del agente y el enemigo, determina la acción que el agente debe tomar y calcula la siguiente posición en función de la acción seleccionada.

```
public CellInfo GetNextStep(CellInfo currentPosition, CellInfo otherPosition)
{
    //Debug.Log("QMindTester: GetNextStep");
    //estado en el que estamos
    currentState = new State(currentPosition, otherPosition, _worldInfo);
    //accion a hacer
    int action = _qTable.GetAction(currentState);
    //siguiente posicion
    CellInfo newAgentPos = _worldInfo.NextCell(currentPosition,
        _worldInfo.AllowedMovements.FromIntValue(action));

    return newAgentPos;
}
```

# Entrenamiento

Para el entrenamiento de nuestro agente hemos probado a ejecutar varios episodios en distintos escenarios. Estos son algunos ejemplos:



A pesar de ello, no se ha conseguido que el agente realice más de 83 pasos. No identificamos si el error reside en los valores de la recompensas, ya que también hemos probado diversos tipos de valores.