Memoria Práctica 2

Desarrollo de juegos con inteligencia artificial

Santiago Rubio Montero
Javier de las Peñas Férnandez

ÍNDICE

INTRODUCCIÓN

El objetivo de la práctica es implementar un agente inteligente cuyo comportamiento sea aprendido por técnicas de machine learning. En concreto, se emplean técnicas de Q-Learning. Para ello se han creado dos archivos C# llamados “*Mytrainer*” en el cual se creará la tabla de exploración a medida que nuestro agente descubre el tablero para posteriormente exportarlo en formato CSV, y “*Mytester*” el cual recibirá la tabla con el aprendizaje y decidirá qué camino es el más óptimo.

MYTRAINER

El objetivo de esta clase es explorar el tablero y calcular todos los valores Q de las celdas por las cuales nuestro agente vaya pasando. Para ello se hace uso de distintas funciones explicadas a continuación.

Initialize:

En primer lugar, declaramos la tabla que nos servirá para rellenar con todos los valores Q correspondientes a cada celda de la tabla. Acto seguido calculamos en la función *Initialize* el número de filas, columnas y rellenamos la tabla con el valor 0 en todas las casillas para posteriormente ser reemplazados por el valor necesario.

```
//Tabla Q
public float[,] tableQ { get; set; }
private int nRows { get; set; }
private int nCols { get; set; }

public void Initialize(QMind.QMindTrainerParams qMindTrainerParams, WorldInfo worldInfo, INavigationAlgorithm navigationAlgorithm)
{
    this.nRows = 4; //Numero de acciones posibles (izquierda, derecha, arriba, abajo)
    this.nCols = worldInfo.WorldSize.x * worldInfo.WorldSize.y; //Cálculo del grid
    this.tableQ = new float[nRows, nCols];

    for (int i = 0; i < this.nRows; i++)
    {
        for (int j = 0; j < this.nCols; j++)
        {
            this.tableQ[i, j] = 0.0f;

        }
    }
```

DoStep:

Después de tener nuestra tabla creada e inicializada toca ir rellenando los valores en función de la exploración de nuestro agente. Para ello hacemos uso de un booleano explorar que nos indicará si podemos explorar o no en caso de haber sido pillados por el player o no. En caso de ser true, y podamos explorar, el player empieza a moverse y el agente también. El agente puede elegir un rango de acciones entre 0 y 4 correspondiente al Norte, Sur, Este u Oeste. Una vez escogida de forma aleatoria la acción, lo movemos y calculamos el estado, que es el número de la celda en la que se ubica este.

```
public void DoStep(bool train, WorldInfo worldInfo)
{
    if (explorar)
    {
        //Movimiento del player (A*)
        CellInfo otherCell = QMind.Utils.MoveOther(_navigationAlgorithm, OtherPosition, AgentPosition);
        OtherPosition = otherCell;

        //Movimiento del agente (Q learning)

        float randomExploration = UnityEngine.Random.Range(0f, 1f);
        int action = UnityEngine.Random.Range(0, 4);
        CellInfo agentCell = QMind.Utils.MoveAgent(action, AgentPosition, worldInfo);
        int state = agentCell.x * worldInfo.WorldSize.y + agentCell.y;
        CellInfo agentcell;
```

Cálculo de acción del agente

Una vez ubicado el estado en el que está, se le da un parámetro aleatorio entre 0 y uno, proporcionado por el ***randomExploration***, si dicho parámetro es menor a Epsilon (Introducido por *parameters.epsilon*), siempre y cuando la celda no sea accesible, se vuelve a dar una acción aleatoria y se mueve el agente y se comprueba otra vez el bucle.

```
if (randomExploration <= parameters.epsilon)
{
    while (!agentCell.Walkable)
    {
        action = UnityEngine.Random.Range(0, 4);
        agentCell = QMind.Utils.MoveAgent(action, AgentPosition, worldInfo);
    }
```

Cálculo de acción del agente si la celda no es accesible

Si es ***walkabkle*** sale del bucle y del if y empieza a calcular los datos de la Q, la recompensa y la máxima Q explorada.

```
float q = getQ(AgentPosition, action, worldInfo);
float reward = GetReward(agentCell, AgentPosition);
float maxQ = GetMaxQ(agentCell, worldInfo);
```

randomExploration > a epsilon

En caso de no ser accesible la acción que calculamos ,se calcula mediante mejorAccionQ(AgentPosition) la casilla con mejor Q de las vecinas y te devuelve la acción que deberías hacer.

```
} else
{
    Debug.Log("NO EXPLORAMOS!");
    action = mejorAccionQ(AgentPosition);
    agentcell = QMind.Utils.MoveAgent(action, AgentPosition, worldInfo);
}
```

randomExploration > a epsilon

Al salir del bucle, después de calcular la correspondiente Q, recompensa y la máxima Q explorada, si el valor de la casilla en la que está el agente es igual a 0, se sobreescribe mediante ***Update_Rule***, que con los parámetros calculados anteriormente calcula la fórmula de la Q.

```
if (tableQ[action, state] == 0.0f)
{
    tableQ[action, state] = Update_rule(q, reward, maxQ);
}
Debug.Log("Valor de q: " + tableQ[action, state]);
AgentPosition = agentCell;
```

Condición para calcular la Q de la casilla

En el momento en el que el player pille al agente, y coincidan en una casilla, se escribe el fichero en la dirección dada.

```
if (OtherPosition == AgentPosition)
{
    File.WriteAllLines("Assets/Scripts/Grupo9/tableQ.csv", ToCsv(tableQ));
}
```

Condición para escribir el fichero CSV

getQ:

Esta función calcula el estado en el que se encuentra el jugador mediante la posición x e y de dicha casilla. Además le añade el valor Q previamente calculado a dicha casilla.

```
public float getQ(CellInfo currentCell, int action, WorldInfo worldInfo)
{

    int estado = (currentCell.x * worldInfo.WorldSize.y) + currentCell.y;
    float qValue = 0.0f;
    qValue = tableQ[action, estado];

    return qValue;

}
```

Función getQ

GetReward:

Esta función calcula la recompensa de la casilla en función de si el agente avanza alejándose o no. En caso de que escoja una acción que lo aleje del player se le recompensa, en caso de que no se le penalice.

```
public float GetReward(CellInfo nextCell, CellInfo currentCell)
{
    if (nextCell.Walkable &&
        nextCell.Distance(OtherPosition, CellInfo.DistanceType.Manhattan) >
        currentCell.Distance(OtherPosition, CellInfo.DistanceType.Manhattan))
    {
        return 100.0f;
    }
    else return -1.0f;
}
```

Función GetReward

Update_rule:

Calcula la función Q con la fórmula dada:

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

```
public float Update_rule(float currentQ, float reward, float maxQ)
{
    float aux = 0.0f;
    aux = (1 - parameters.alpha) * currentQ + parameters.alpha * (reward + parameters.gamma *
   maxQ);
    return aux;
}
```

Función Update_rule

ToCsv<>:

Escribe el fichero en formato CSV mediante un StreamWriter en una dirección concreta. Para ello lo hace mediante un bucle que recorre la tabla casilla por casilla escribiendo sus valores.

```
1 referencia
private static IEnumerable<String> ToCsv<T>(T[,] data, string separator = "/")
{
    for (int i = 0; i < data.GetLength(0); ++i)
        yield return string.Join(separator, Enumerable
          .Range(0, data.GetLength(1))
          .Select(j => data[i, j])); // simplest, we don't expect ',' and '"' in the items
}
```

Función IEnumerable

MYTESTER

Esta clase se encarga de explotar la información aprendida por la tabla Q en el modo entrenamiento, para ello, se carga la tabla desde un csv previamente guardado y se separan los datos, que están divididos entre barras. Para esta tarea se ha implementado el método loadTableQ, el cual recorriendo con un bucle las posiciones de la tabla, la copia en una nueva creada para poder ser leída desde esta clase.

```
1 referencia
private float[,] LoadQTable(float[,] table)
{
    if (File.Exists(filePath))
    {
        // Lee todas las líneas del archivo CSV
        string[] lines = File.ReadAllLines(filePath);

        for (int i = 0; i < lines.Length; i++)
        {
            // Divide cada línea en valores usando la coma como separador
            string[] values = lines[i].Split('/');

            for (int j = 0; j < values.Length && j < nCols; j++)
            {
                // Convierte cada valor a float y almacénalo en la matriz
                float parsedValue;
                if (float.TryParse(values[j], out parsedValue))
                {
                    table[i, j] = parsedValue;
                }
                else
                {
                    Debug.LogError("Error al analizar el valor en la posición [" + i + ", " + j + "]");
                }
            }
        }

        // La matriz bidimensional 'table' ahora contiene los valores del CSV
        Debug.Log("CSV leído exitosamente");

        // Puedes asignar 'table' a 'tableQ' si es necesario
        tableQ = table;
        return tableQ;
    }
    else
    {
        Debug.LogError("El archivo CSV no existe en la ruta: " + filePath);
    }
    return null;
}
```

Función LoadQTable

ANEXOS:

Somos conscientes de que distintos elementos de la práctica no funcionan como deberían. En primer lugar el cálculo de estados es mejorable ya que se deberían haber conformado los estados entre los obstáculos, o casillas non Walkable y la distancia del enemigo con su respectiva posición.Esto hubiese hecho que se obtengan muchos menos estados y hubiesen sido mas genéricos, ya que de la forma que hemos calculado, los estados corresponden a un estado por cada posición absoluta del agente y por otra parte del Player. En segundo lugar la clas *MyTester*, puede leer la tabla *Q* previamente creada y guardarla correctamente, sin embargo a la hora de decidir la dirección a la que se dirige el personaje, no sabemos por qué no funciona. Intuimos que tiene relación con la forma en que hemos calculado los estados.