Memoria de la práctica 2

Desarrollo de un Modelo de Aprendizaje por Refuerzo con Q-Learning

Universidad
Rey Juan Carlos

Adrián Espínola Gumiel

Antonio Machorro Herrera

David Antonio Paz Gullón

1. Contexto del juego

El entorno del juego simula un escenario en el que un agente debe huir de un jugador que intenta capturarlo en un espacio delimitado. El agente y el jugador interactúan en un tablero dividido en celdas, algunas de las cuales pueden ser inaccesibles (paredes). El objetivo principal del agente es maximizar su supervivencia evitando al jugador y utilizando una estrategia óptima basada en su aprendizaje previo.

Este escenario ofrece un problema ideal para aplicar técnicas de **aprendizaje por refuerzo** debido a la interacción dinámica entre el agente, el jugador y el entorno. Específicamente se emplea el método de Q-Learning para generar una tabla de valores Q que le regresan al agente la mejor dirección posible en un estado dado.

2. Problema planteado a la IA

El desafío que enfrenta la inteligencia artificial (IA) consiste en:

Decidir las mejores acciones en cada estado del juego:

- Moverse hacia celdas accesibles.
- Evitar al jugador manteniendo una distancia segura.

Maximizar las recompensas acumuladas:

- Recibir penalizaciones por decisiones erróneas, como moverse hacia celdas no transitables o ser capturado por el jugador.
- Obtener recompensas al aumentar la distancia con el jugador.

La IA debe aprender a balancear entre exploración (probar nuevas acciones) y explotación (utilizar lo aprendido) para alcanzar su objetivo de manera óptima.

3. Conceptos Específicos del Algoritmo

El algoritmo implementado utiliza **Q-Learning**, una técnica de aprendizaje por refuerzo basada en la actualización de valores Q asociados a cada combinación de estado y acción. A continuación, se describen los elementos clave:

- **Estado del Juego**: Cada estado representa una configuración específica del entorno, modelada por:
    - Paredes presentes en las **direcciones cardinales** (norte, sur, este, oeste).
    - **Distancia de Manhattan** al jugador.
    - Posición relativa del jugador respecto al agente.
- **Acciones**: El agente puede realizar cuatro acciones:

- Moverse hacia arriba.
- Moverse hacia abajo.
- Moverse hacia la derecha.
- Moverse hacia la izquierda.

- **Tabla Q (QTable)**: Una estructura que almacena valores Q, que representan la calidad esperada de realizar una acción específica en un estado dado. Estos valores se actualizan durante el entrenamiento utilizando la fórmula:

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

Donde:

- α: Tasa de aprendizaje.
- γ: Factor de descuento.
- r: Recompensa recibida al realizar la acción a.
- maxQ(s′,a): Mejor valor Q esperado para el siguiente estado s′.

4. Metodología

Se utiliza una arquitectura modular que incluye:

1. **QTraining**: encargado del proceso de entrenamiento.
2. **QTable**: una estructura para almacenar los valores Q.
3. **State**: que modela las características del entorno a través de un estado.
4. **QTester**: para evaluar la capacidad del modelo entrenado en escenarios controlados.

5. Estrategia de Recompensas

- **Recompensas Positivas**:
  - Aumentar la distancia al jugador: +5 · (diferencia de distancia)

    La única recompensa positiva que se ha otorgado es cuando el agente logra aumentar la distancia entre él y el jugador.
- **Penalizaciones**:
  - Mantener la misma distancia: −5.
  - Reducir la distancia al jugador: −15.

- Ser capturado por el jugador: −200.
- Intentar moverse hacia una celda no transitable: −999.

Cuando el agente no aumenta su distancia, es capturado por el jugador o intenta pasar a una casilla inaccesible, se le penaliza dependiendo del tamaño del error.

6. Funcionamiento de los scripts:

- **QTraining:**

El script QTraining implementa la funcionalidad principal del entrenamiento del agente mediante el uso de la tabla Q (QTable). Su lógica se organiza en torno a los episodios y pasos. Los aspectos clave incluyen:

1. Inicialización:
    - Se definen el número de acciones posibles (4) y los estados posibles (calculados como combinaciones de paredes, distancia de Manhattan y posición relativa).
    - Si existe una tabla Q previa, esta se carga; de lo contrario, se crea una nueva.

```
public void Initialize(QMindTrainerParams qMindTrainerParams, WorldInfo worldInfo, INavigationAlgorithm navigationAlgorithm)
{
    _params = qMindTrainerParams;
    _worldInfo = worldInfo;
    _navigationAlgorithm = navigationAlgorithm;
    _navigationAlgorithm.Initialize(worldInfo);

    int possibleActions = 4;
    int possibleStates = 16 * 5 * 39;

    string filePath = @"Assets/Scripts/GrupoT/QTable.csv";
    if (File.Exists(filePath))
    {
        Debug.Log("Loading existing QTable...");
        _qTable = new QTable(possibleActions, possibleStates);
        LoadQTable(filePath);
    }
    else
    {
        Debug.Log("Creating new QTable...");
        _qTable = new QTable(possibleActions, possibleStates);
    }

    AgentPosition = worldInfo.RandomCell();
    OtherPosition = worldInfo.RandomCell();

    CurrentEpisode = 0;
    CurrentStep = 0;
    Return = 0;
    ReturnAveraged = 0;

    OnEpisodeStarted?.Invoke(this, EventArgs.Empty);

    Debug.Log("QMindTrainerDummy: initialized");
}
```

2. Gestión de episodios:
    - Al inicio de cada episodio, el agente y el jugador se colocan en posiciones aleatorias. Los pasos se reinician a cero.

- Se define un manejador de eventos para señalar el inicio y fin de cada episodio.

```
    if (OtherPosition == null || CurrentStep == _params.maxSteps || OtherPosition == AgentPosition)
    {
        OnEpisodeFinished?.Invoke(this, EventArgs.Empty);
        NewEpisode();
    }
    else
    {
        CurrentStep++;
    }

    Debug.Log($"Agent moved to {AgentPosition.x}, {AgentPosition.y} using action {action}");
}

private void NewEpisode()
{
    CurrentEpisode++;

    AgentPosition = _worldInfo.RandomCell();
    OtherPosition = _worldInfo.RandomCell();

    CurrentStep = 0;
    Return = 0;

    if(CurrentEpisode % _params.episodesBetweenSaves == 0)
    {
        SaveQTable(@"Assets/Scripts/GrupoT/QTable.csv");
    }

    OnEpisodeStarted?.Invoke(this, EventArgs.Empty);

    Debug.Log($"New episode started. Episode: {CurrentEpisode}, Agent: ({AgentPosition.x}, {AgentPosition.y}), Player: ({OtherPosition.x}, {OtherPosition.y})");
}
```

3. Ejecución de pasos:
    - El agente toma decisiones basadas en la exploración (acciones aleatorias) o explotación (mejor acción según la tabla Q). La decisión que toma se calcula utilizando el parámetro de exploración *épsilon*.
    - Si el agente intenta moverse a una celda no transitable, se penaliza (-999) y se repite el bucle hasta que encuentre una posición válida.
    - Se calcula la recompensa según la distancia inicial y final al jugador:
        - Si el agente se acerca al jugador, recibe penalizaciones (-200 si es capturado).
        - Si se aleja, obtiene una recompensa positiva proporcional a la distancia.
        - Mantenerse estático resulta en una penalización leve (-5).

```
private float CalculateReward(CellInfo nextCell)
{
    float iDistance = AgentPosition.Distance(OtherPosition, CellInfo.DistanceType.Manhattan);
    float fDistance = nextCell.Distance(OtherPosition, CellInfo.DistanceType.Manhattan);

    if (nextCell == OtherPosition) return -200;
    if (fDistance > iDistance) return 5 * (fDistance - iDistance);
    if (fDistance < iDistance) return -15;
    return -5;
}
```

4. Actualización de la tabla Q:
    - Se aplica la fórmula de actualización basada en los parámetros de aprendizaje (α, γ) para ajustar los valores Q.

```
public float UpdateQ(float currentQ, float reward)
{
    float bestNextQ = _qTable.GetBestQ(GetCurrentStateId());
    Debug.Log($"The calculation will be: currentQ: {currentQ}, reward: {reward}, bestNextQ:{bestNextQ}");
    return (1 - _params.alpha) * currentQ + _params.alpha * (reward + _params.gamma * bestNextQ);
}
```

5. Guardado de la tabla Q:
    - Cada ciertos episodios, la tabla se guarda en un archivo CSV, donde los estados se representan con sus características y valores Q.

```
private void NewEpisode()
{
    CurrentEpisode++;

    AgentPosition = _worldInfo.RandomCell();
    OtherPosition = _worldInfo.RandomCell();

    CurrentStep = 0;
    Return = 0;

    if(CurrentEpisode % _params.episodesBetweenSaves == 0)
    {
        SaveQTable(@"Assets/Scripts/GrupoT/QTable.csv");
    }

    OnEpisodeStarted?.Invoke(this, EventArgs.Empty);

    Debug.Log($"New episode started. Episode: {CurrentEpisode}, Agent: ({AgentPosition.x}, {AgentPosition.y}), Player: ({OtherPosition.x}, {OtherPosition.y})");
}
```

- **QTable:**

La clase QTable gestiona la estructura de la tabla Q, permitiendo almacenar y actualizar valores Q para cada estado y acción.

1. Estados:
    - Los estados se generan considerando la presencia de paredes, la distancia de Manhattan al jugador y la posición relativa.
    - Se inicializan más de 3,000 combinaciones posibles, asegurando que todos los estados posibles estén predefinidos.

```
public void InitializeStates()
{
    int stateId = 0;
    for (int n = 0; n <= 1; n++)
    {
        for (int s = 0; s <= 1; s++)
        {
            for (int e = 0; e <= 1; e++)
            {
                for (int w = 0; w <= 1; w++)
                {
                    for (int pos = 0; pos <= 4; pos++)
                    {
                        for (int dist = 0; dist <= 38; dist++)
                        {
                            State state = new State(n == 1, s == 1, e == 1, w == 1, dist, pos, stateId);
                            AddState(state);
                            stateId++;
                        }
                    }
                }
            }
        }
    }

    Debug.Log($"Initialized {_stateList.Count} states.");
}
```

2. Valores Q:
    - Los valores iniciales de Q se establecen en cero.

- Métodos como GetQValue y UpdateQValue permiten obtener y actualizar valores para un estado y acción específicos.

```
public float GetQValue(int stateId, int action)
{
    int stateIndex = GetStateIndex(stateId);
    return _qTable[stateIndex, action];
}

public void UpdateQValue(int stateId, int action, float newQ)
{
    int stateIndex = GetStateIndex(stateId);
    Debug.Log($"Updated Q value at: {stateIndex} with an action of {action} and a new Q of: {newQ}");
    _qTable[stateIndex, action] = newQ;
}
```

3. Selección de acciones:

    - GetBestAction identifica la acción con el mejor valor Q, mientras que GetBestQ devuelve el mejor valor para un estado dado.

```
public float GetBestQ(int stateId)
{
    int stateIndex = GetStateIndex(stateId);
    float maxQ = float.MinValue;

    for (int i = 0; i < _actionNum; i++)
    {
        if (_qTable[stateIndex, i] > maxQ)
        {
            maxQ = _qTable[stateIndex, i];
        }
    }

    return maxQ;
}

public int GetBestAction(int stateId)
{
    int stateIndex = GetStateIndex(stateId);
    float maxQ = float.MinValue;
    int bestAction = -1;

    for (int i = 0; i < _actionNum; i++)
    {
        if(_qTable[stateIndex, i] > maxQ)
        {
            maxQ = _qTable[stateIndex, i];
            bestAction = i;
        }
    }

    return bestAction;
}
```

- **State:**

La clase State define las características de un estado:

- Presencia de paredes en las direcciones cardinales.
- Distancia de Manhattan al jugador.
- Posición relativa al jugador.
- Identificador único del estado (stateId).

Esta clase es fundamental para identificar y manejar los estados en el contexto de la tabla Q.

```
public class State
{
    public bool NorthIsWall { get; set; }
    public bool SouthIsWall { get; set; }
    public bool EastIsWall { get; set; }
    public bool WestIsWall { get; set; }
    public float ManhattanDistanceToPlayer { get; set; }
    public int PositionToPlayer { get; set; }
    public int stateId { get; set; }

    public State(bool nIsWall, bool sIsWall, bool eIsWall, bool wIsWall, float manhattan, int position, int stateId)
    {
        NorthIsWall = nIsWall;
        SouthIsWall = sIsWall;
        EastIsWall = eIsWall;
        WestIsWall = wIsWall;
        ManhattanDistanceToPlayer = manhattan;
        PositionToPlayer = position;
        this.stateId = stateId;
    }
}
```

- **QTester:**

El script QTester replica el comportamiento de QTraining, pero con el propósito de probar el modelo ya entrenado en lugar de seguir entrenándolo. Sus principales funciones incluyen:

1. Carga de la tabla Q:
    - Carga los datos desde el archivo CSV generado durante el entrenamiento.

```
public void LoadQTable(string filePath)
{
    if (!File.Exists(filePath))
    {
        Debug.LogError($"QTable not found {filePath}");
        return;
    }

    string[] lines = File.ReadAllLines(filePath);
    for (int i = 1; i < lines.Length; i++)
    {

        string line = lines[i].Trim();
        if (string.IsNullOrEmpty(line)) continue;

        string[] values = line.Split(',');

        if (values.Length < 11)
        {
            Debug.LogError($"Malformed line: {line}");
        }

        try
        {

            int stateId = int.Parse(values[0]);

            if (stateId >= 16 * 5 * 39)
            {
                Debug.LogError($"Invalid stateId: {stateId}. Skipping line: {line}");
                continue;
            }

            for (int action = 0; action < 4; action++)
            {
                float qValue = float.Parse(values[7 + action]);
                _qTable.UpdateQValue(stateId, action, qValue);
            }
        }
        catch (Exception ex)
        {
            Debug.LogError($"Error parsing line: {line}. Exception: {ex.Message}");
        }
    }

    Debug.Log($"QTable loaded. {_qTable.GetBestQ(87)}");
}
```

2. Selección de acciones:
    - Para cada paso, identifica el estado actual basado en las condiciones del entorno y calcula la mejor acción a tomar según los valores Q.
3. Movimiento del agente:

- Devuelve la dirección calculada al método GetNextStep() para mover al agente.

```
public CellInfo GetNextStep(CellInfo currentPosition, CellInfo otherPosition)
{
    try
    {
        bool northIsWall = !_worldInfo.NextCell(currentPosition, Directions.Up).Walkable;
        bool southIsWall = !_worldInfo.NextCell(currentPosition, Directions.Down).Walkable;
        bool eastIsWall = !_worldInfo.NextCell(currentPosition, Directions.Right).Walkable;
        bool westIsWall = !_worldInfo.NextCell(currentPosition, Directions.Left).Walkable;

        float manhattanDistance = currentPosition.Distance(otherPosition, CellInfo.DistanceType.Manhattan);
        int positionToPlayer = CalculatePosition(currentPosition, otherPosition);

        State currentState = new State(northIsWall, southIsWall, eastIsWall, westIsWall, manhattanDistance, positionToPlayer, -1);

        Debug.Log($"Current state: N={currentState.NorthIsWall}, S={currentState.SouthIsWall}, E={currentState.EastIsWall}, W={currentState.WestIsWall}, Dist={currentState.ManhattanDistanceToPlayer}, Pos={currentState.PositionToPlayer}");
        int stateId = GetStateId(currentState);
        Debug.Log($"State id: {stateId}");
        Debug.Log($"Best Q:{_qTable.GetBestQ(stateId)}");

        float bestQ = _qTable.GetBestQ(stateId);

        int bestAction = _qTable.GetBestAction(stateId);

        Debug.Log($"Best action: {bestAction}");

        Directions direction = GetDirection(bestAction);
        return _worldInfo.NextCell(currentPosition, direction);
    }
    catch (Exception ex)
    {
        Debug.LogError($"Error in GetNextStep: {ex.Message}");
        return null;
    }
}
```

7. Resultados y Funcionalidad

El agente, tras un proceso de entrenamiento de múltiples episodios, muestra un desempeño eficiente, alcanzando más de 5000 pasos en escenarios complejos. La combinación de aprendizaje autónomo, penalizaciones y recompensas permite al agente desarrollar estrategias para sobrevivir en un entorno dinámico y hostil.

El código es completamente funcional y puede ejecutarse dentro del proyecto base sin errores, demostrando la aplicabilidad y efectividad del aprendizaje por refuerzo en entornos simulados.