Practica sobre Machine Learning

Pertenecemos a un estudio de videojuegos que pretende desarrollar un juego de tanques similar a Battle City, un clasico de la NES. Para ello se ha desarrollado un prototipo jugable que teneis a vuestro disposición en el campus virtual.

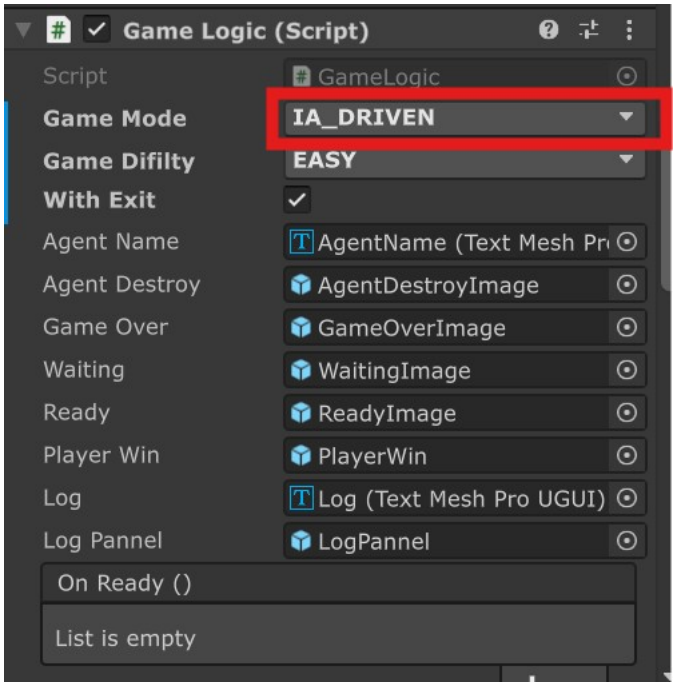
Vuestro cometido es desarrollar una IA para el jugador. Esta IA la van a utilizar para suplir al jugador en un mecanismo de testing automático. Por lo tanto, debeis entrenar a la IA para que juegue al juego por vosotros.

Ejercicio 1 Juega al juego varias veces:

En el proyecto de Unity adjunto, cargando **SceneImitation01** tenemos el nivel donde entrenaremos a nuestra IA. En concreto en **BasicLogic** En el componente **GameLogic** podemos cambiar el Game Mode a:

- Normal: Para probar el juego
- Recorded: Para guardar las partidas jugadas en un csv
- IA_Driven: Para ejecutar el comportamiento aprendido desde Unity.

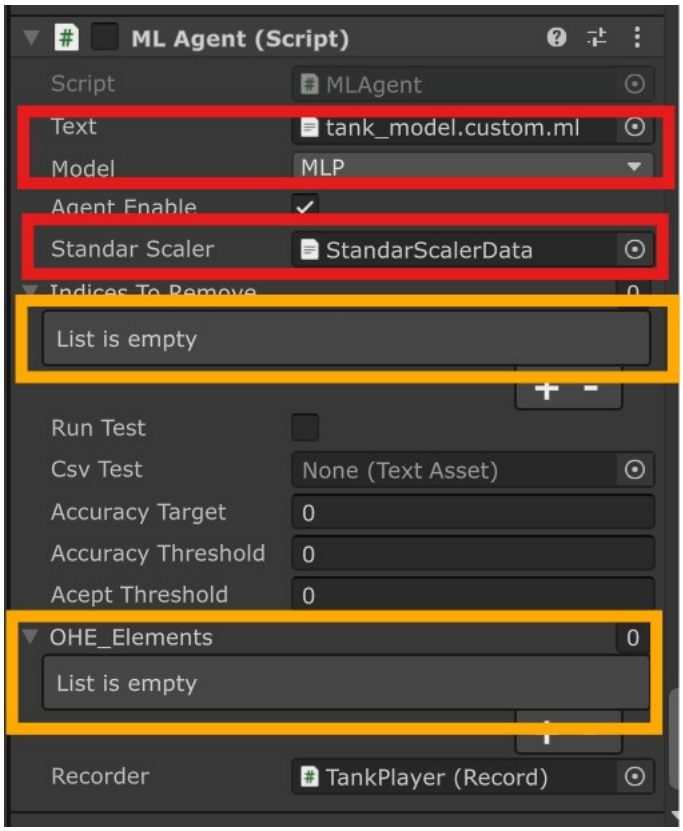
Tambien hay dos modos de dificultad, pero la práctica la realizaremos con el Easy como primera opción, dejando la opción de hacerlo en normal para los más aventureros y aventurares :)



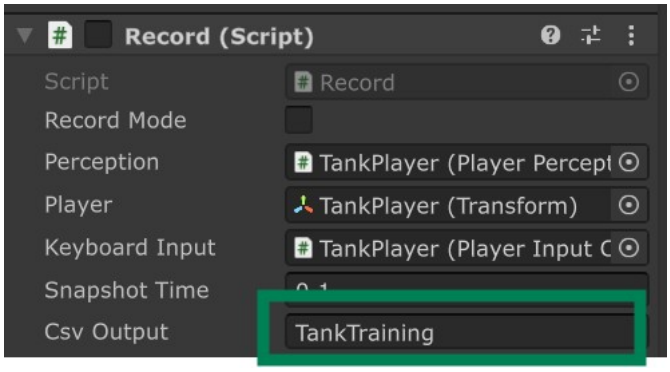
En el prefab de **TankPlayer** en **MLAgent** podemos configurar algunas cosas. En text (en rojo) tenemos el modelo exportado que entrernaremos en Python.

En Standar Scaler tendremos las medias y desviaciones que nos porporcionará Python con las que hemos escalado para que el input este en el mismo formato que el procesado para crear el modelo.

Tambien en amarillo tenemos los indices de las variables a las que hayamos hecho One Hot Encoding, para aplicarlo en el juego.



En este mismo prefab tenemos el componente Record



Que se encarga de grabar las trazas.

Describamos algunas clases mas que son interesantes.

- **PlayerPerception**: es una clase que implementa la percepción del agente. Podéis modificarla todo lo que queráis para poder añadir más información al estado del juego. Pero por defecto estas las features que tiene el modelo (Más el tiempo que es la última):

```
public enum Player_PARAMETER_ID
{
    NEIGHBORHOOD_UP = 0, NEIGHBORHOOD_DOWN = 1,
    NEIGHBORHOOD_RIGHT = 2, NEIGHBORHOOD_LEFT = 3,

    NEIGHBORHOOD_DIST_UP = 4, NEIGHBORHOOD_DIST_DOWN = 5, NEIGHBORHOOD_DIST_RIGHT = 6,

    NEIGHBORHOOD_DIST_LEFT = 7,
    COMMAND_CENTER_X = 8, COMMAND_CENTER_Y = 9,
    AGENT_1_X = 10, AGENT_1_Y = 11,
    AGENT_2_X = 12, AGENT_2_Y = 13,
    CAN_FIRE = 14, HEALTH = 15, LIFE_X = 16,LIFE_Y = 17,
    EXIT_X = 18,EXIT_Y = 19

}
```

- NEIGHBORHOOD_XXX: es el ID del vecino más cercano en las 4 direcciones principales. Los ids son los siguientes (**PerceptionBase**)

```
public enum INPUT_TYPE { NOTHING = 0, UNBREAKABLE = 1, BRICK = 2,
COMMAND_CENTER = 3, PLAYER = 4,
SHELL = 5, OTHER_AGENT = 6, LIFE = 7,
SEMI_BREKABLE = 8, SEMI_UNBREKABLE = 9, EXIT = 10 }
```

- NEIGHBORHOOD_DIST_XXX: es la distancia a la que se encuentra el objeto detectado en cada dirección
- COMMAND_CENTER_X/Y: posición a la command center
- AGENT_1_X/Y posicion del agente 1 (100 si ha sido eliminado)
- AGENT_2_X/Y posicion del agente 2 (100 si ha sido eliminado)
- CAN_FIRE: Si puede disparar en este momento. Es un parametro que peude servirnos para predecir el disparo pero el agente disparará automanticamente en su modo inicial por simplificar el problema a aprender.
- HEALTH: LA vida que le queda al player
- LIFE_X/Y la posición del item de vida
- EXIT_X la posición del iten de fin de juego (puerta)

Elementos en el juego:

- El iten de vida (con un tanque) te aumenta la vida en 1 unidad.
- El iten de salida (estrella) solo está activo si los dos agentes han sido destruidos.

- Command Center: es lo que hay que defender. Si alguien la rompe se acaba el juego y pierdes. También pierdes si el jugador pierde todas sus vidas.
- No se puede disparar una bala mientras hay una disparada.

Las acciones posibles son (**PerceptionBase**):

```
public enum ACTION_TYPE { NOTHING = 0, MOVE_UP = 1,
MOVE_DOWN = 2, MOVE_RIGHT = 3, MOVE_LEFT = 4 }
```

La idea es jugar varias veces con el modo **Record** seleccionado. El juego es complejo, por lo que se requieren de varias partidas. Cuanto mas variadas sean las partidas mas casos se le enseña al entrenamiento. Recordad que lo importante son los datos para que los modelos aprendan bien. Si solo le damos datos sesgados no podrán aprender gran cosa.

Ejercicio 2 visualiza los datos (1 punto):

Crea un CSV con todas las trazas del juego que hayas generado y llévatelo a Python / Jupiter Notebook.

Ahí dibuja la distribución de clases que hayáis grabado con una gráfica que las posicione en un espacio 2D o 3D, con una clase que sea representada con un color o forma diferente de la de otra. Ten cuidado porque tendrá seguramente más de 3 atributos y no podréis visualizarlo directamente. Consulta como hacerlo en el Tema 6 de Data Mining.

Ejercicio 3 Limpia el dataset (1 punto):

Crea una versión del dataset limpia con la información que sea relevante tanto en atributos usados como en acciones. Comprueba que no haya valores erroneos o que no tengan sentido y corrígelos.

También en esta sección debes normalizar los datos usando StandarScaler o aplicar OneHotEncoding o Label/Ordinal encoder donde sea necesario.

Ejercicio 4 Prueba diferentes modelos de Machine Learning (hasta 5 puntos):

- Al menos uno de ellos debe ser vuestra implementación del perceptrón multicapa. El perceptrón multicapa debe permitir crear modelos de cualquier número de capas. Al menos en el jupiter notebook a entregar debe haber un ejemplo con más de 3 capas para poder contar con el punto asociado con esta característica, independientemente que el mejor modelo resulta ser con una sola capa oculta. **(1 punto)**
- Comprueba que los resultados de tu implementación son similares (que no idénticos) al MLPClassifier de SKLearn (poner los mismos valores de alpha, learning_rate_init y de **usar la función logística o sigmoidal**

para las capas ocultas) El resultado de validación debe ser superior al 80% par obtener el punto **(1 punto)**

- A parte de esta versión, podéis cacharrear con diferentes parámetros de la versión de SKLearn del MLPClassifier (con función de activación relu o con diferentes versiones del optimizador) hasta conseguir un buen resultad de validación. El resultado de validación debe ser superior al 80% para obtener el punto **(1 punto)**
- Crea un modelo KNN y comprueba el resultado que has obtenido. Intenta que el modelo tenga un resultado lo mas similar o superior al perceptrón que puedas. **(1 punto)**
- Crea un modelo de árboles de decisión y otro de Random Forest y comprueba el resultado que has obtenido. Intenta que los modelos tengan un resultado lo mas similar o superior al perceptrón que puedas. **(1 punto)**
- Muestra las matrices de confusión de todos los modelos usados, también calcula accuracy y otras métricas para todos los modelos y al final en el propio Notebook usando Markdown explica que modelo crees que se adapta mejor al juego y cual elegirías.
- Itera entre los modelos, sus hiperparámetros, los datos exportados y la limpieza de los mismos, así como el número de ejemplos de entrenamiento hasta conseguir modelos con el mejor rendimiento teórico posible.

Ejercicio 5 Implementa el perceptrón multicapa que quieras en Unity (2 puntos).

Puedes elegir vuestro modelo o el modelo de MLPClassifier de SKLearn. Por defecto viene la fontanería necesaria para poder implementar el modelo de MLPClassifier en Unity, asi que te recomiendo que uses ese, pero si te animas puedes usar el tuyo. Para poder exportar dicho modelo a diferentes formatos podéis usar el método **Utils.ExportAllformatsMLPSKlearn** que os exporta el modelo a diferentes formatos: pickle, onnx, JSON y un formato custom que os facilita poder exportar vuestro modelo si así lo consideráis. Unity utiliza esta versión custom para cargar el modelo. **NOTA exportad el modelo con la función logistica como función de activación si usais la libreria**

El formato custom se comporta de la siguiente forma (cada campo es una línea):

```
num_layers:4
parameter:0
dims:['9', '8']
name:coefficient
values:[4.485004762391437, 0.7937380313502955, ... -1.360755118312314]
parameter:0
dims:['1', '8']
name:intercepts
```

```
values:[-0.6982858709618917, 3.3853548406875134,... 0.1310577892518341]
--- repetir por cada capa / theta, cambiando el número de parameter
y coefficient e intercepts tendrán un número asociado por ejemplo coefficient1, coefficient2
```

- num_layer: nos indica le número de capas del perceptrón.
- dims: es la dimensión de la matriz
- name: es el nombre del coeficiente. Las matrices theta se llaman coefficient y los sesgos intercepts. Este es el formato por defecto, pero podéis cambiarlo si vuestro código introduce los sesgos dentro de las matrices, aunque tendréis que cambiar código en Unity.

La implementación actual lee correctamente los parámetros de MLPClassifier de SKLearn. Para conseguir los puntos de este ejercicio mínimo habría que usar esa implementación, pero se valorará usar vuestra propia implementación en el agente. Como podéis ver, SKLearn implementa los sesgos como un vector aparte y no dentro de la matriz de pesos.

Para implementar el agente hay que escribir las siguientes funciones en C# en el componente **MLAgent**:

```
public PerceptionBase.ACTION_TYPE AgentInput()
{
    int action = -1;
    switch (model)
    {
        case ModelType.MLP:
            action = 0;
            //TODO leer de los parámetros de la percepción.
            //Debe respetar el mismo orden que los datos.
            //TODO Llamar a RunFeedForward
            //guardar la toma de decisiones y despues validar si son correctas.
            recorder.AIRecord(action);
            break;
    }
    PerceptionBase.ACTION_TYPE input = Record.ConvertLabelToInput(action);
    return input;
}


public float[] RunFeedForward(float[] modelInput)
{
    //permite eliminar columnas de la percepción si las habeis eliminado en el modelo.
    modelInput = modelInput.Where((value, index) =>
    !indicesToRemove.Contains(index)).ToArray();
    //TODO Hacer las transformaciónes necesarias para ejecutar el modelo

    //Guardamos el model input con las trasformaciones para poder ejecutarlo desde payth
    recorder.AIRecord(modelInput);
```

```
        float[] outputs = this.mlpModel.FeedForward(modelInput);

        return outputs;
    }
```

- **MLP**: Es la implementación de la inferencia de un Perceptrón Multicapa. Sólo hay que programar el FeedForward de la red ya que se entrenará en Python. Aqui hay varios TODOS que teneis que completar (Nota, en iterativo):

```
    public float[] FeedForward(float[] input)
    {
        //TODO: implement feedworward.
        //the size of the output layer depends on
        //  what actions you have performed in the game.
        //By default it is 7 (number of possible
        //  actions) but some actions may not have been performed and therefore the model ha
        //  assumed that they do not exist.
        return new float[5];
    }

    private float sigmoid(float z)
    {
        //TODO implementar
        return 0f;
    }

    public float[] SoftMax(float[] zArr)
    {
        //TODO implementar
        return zArr;
    }

    public int GetIndexMaxValue(float[] output, out float max)
    {
        max = output[0];
        int index = 0;
        //TODO impleemntar.
        return index;
    }
```

Además de lo anterior hay que implementar en **OneHotEncoding**

```
    public float[] Transform(float[] input)
    {
        List<float> output = new List<float>();
        for (int i = 0; i < input.Length; i++)
        {
```

```
            //TODO implementar el OHE.
        }
        return output.ToArray();
    }
```

Y en el **StandarScaler**:

```
    public float[] Transform(float[] a_input)
    {
        float[] scaled = new float[a_input.Length];
        //TODO implementar

        return a_input;
    }
```

Lo ideal es conseguir un modelo que permita **al tanke acabar en la mayoria de las ocasiones con los Agentes.**

Para exportar los datos se proporciona el fichero Utils con ciertas utilidades interesantes. Es posible que tengais que instalar los paquetes:

```
!pip install skl2onnx
!pip install -U onnx2json
```

ExportAllformatsMLPSKlearn exporta a multiples formatos de redes de nueronas.

```
def ExportAllformatsMLPSKlearn(mlp,X,picklefileName,onixFileName,jsonFileName,customFileName
    return
```

WriteStandardScaler es capaz de generar un lista con las medias y desviaciones para cargarla en Unity

```
def WriteStandardScaler(file,mean,var):
    return
```

Ejercicio 6 Crear un modelo que permita al tanque pasarse el nivel

Probablemente tendrás que añadir mas características, o porcesarlas para enviarlas en una ventana temporal. Aqui hay que remangarse para conseguirlo.

Ejercicio 7 Implementa otro ejecutor de un modelo de ML de los que hayas usado en Unity (opcional + 1 punto) o bien un modelo de Deep Learning.

Realizar este apartado aporta **un punto extra** a la nota de la práctica. Si la práctica de por si tiene un 10 o un 9 y pico, el resto de la puntuación se sumará a la puntación del examen.

Puedes elegir cualquier otro ejecutor de modelo que quieras, Decision Tree, KNN o modelos más complejos con deep learning. Si vas a utilizar modelos mas complejos te recomendamos usar ML-Agents o Sentis (Inference Engine), pero si lo que quieres es implementar algo sencillo, puedes hacerlo con modelos como KNN o decisión tree donde crear un ejecutor es algo más o menos fácil de hacer.

Si realizas este apartado, explica en el Notebook si ha funcionado mejor en la práctica que el modelo de MLP que hayáis implementado en el ejercicio anterior. La implementación obviamente debe conseguir un modelo razonablemente bueno para que cuente el punto extra.