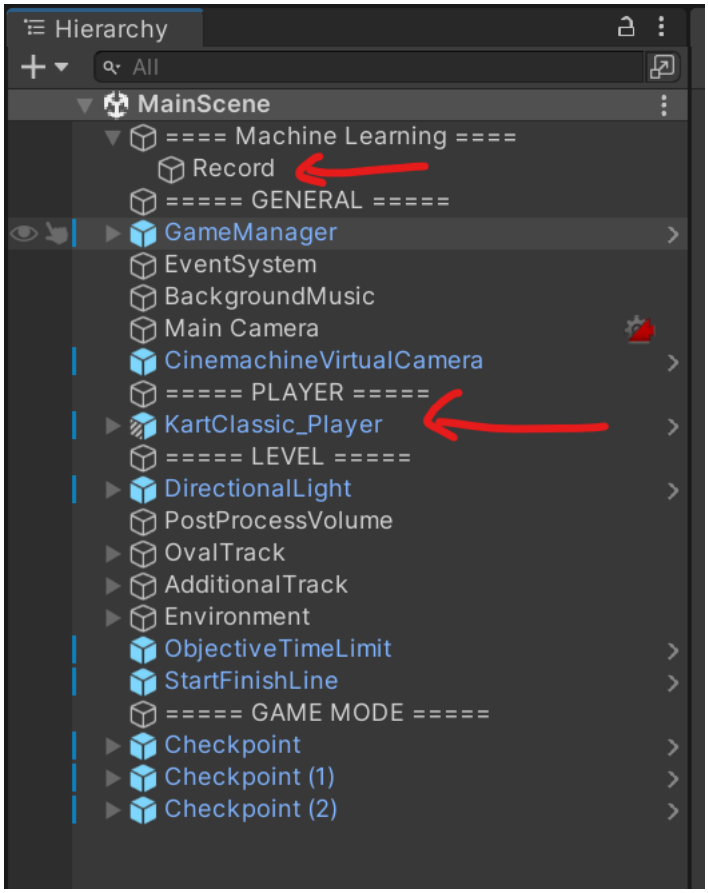
# Practica sobre Machine Learning

Pertenecemos a un estudio de videojuegos que pretende desarrollar un juego de Karts similar a Super Mario Kart. Para ello se ha desarrollado un prototipo jugable que teneis a vuestro disposición en el campus virtual.

Vuestro cometido es desarrollar una IA para los coches controlados por el juego. Para ello, vamos a entrenar a un agente usando jugando al juego multiples veces. Una vez tengamos guardado los datos de ejemplo, nuestro cometido es realizar un estudio teórico sobre que modelo de Machine Learning es más eficiente para implementar el agente.

## Ejercicio 1 Juega al juego varias veces:

En el proyecto de Unity adjunto, cargando MainScene podemos ver la siguiente jerarquia:




El GameObject KartClassic_Player tiene la implementación del control del

coche. Se ha añadido dos componentes, Perception y MLAgent:

- Perception: es una clase que implementa la percepción del agente. Podeis modificarla todo lo que querais para poder añadir mas información al estado del juego. Pero por defecto peude generar un numero de rayos fisicos indicados en el campo Rays de una distancia indicado en Distance. La layer contra la que choca es la 11, que es la layer de la pista. Hay un campo, HeightOffest que nos permite levantar un poco el rayo para que no choque contra la carretera si no contra los bordes de la misma.
- MLAgent: nos permite implmeentar un modleo de Machine Learning. Esta preparado para poder implementar varios modelos, pero por defecto solo implementa parcialmente el MLPClassifier de SkLearn. Cuanto esta activo el check AgentEnable, el player leerá el imput del MLAgent y no del input. En text tenemos el asset con el modelo serializado del MLPClassifier previamente generado desde Python.


El GameObject Record nos permite grabar una partida. Si tenemos el check RecordMode activo al iniciar una partida se guarda la ejecución. El nombre del fichero se indica en Csv Output y se guarda automáticamente si tienes activo el check Save When Finish. Tambien peudes iniciar la grabación y finalizarla pulsando la tecla R.

Perception Names es el nombre que le damos a los campos de los rayos. Además de estos guardamos: - kartx: posición x del Kart. - karty: posición y del Kart. - kartz: posición z del Kart. - time: tiempo en segundos donde se toma la captura de datos. - action: acción que ha realizado el jugador.

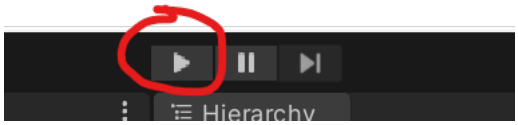
Las acciones posibles son:

```
public enum Labels { NONE=0, ACCELERATE=1, BRAKE=2, LEFT_ACCELERATE=3,
```

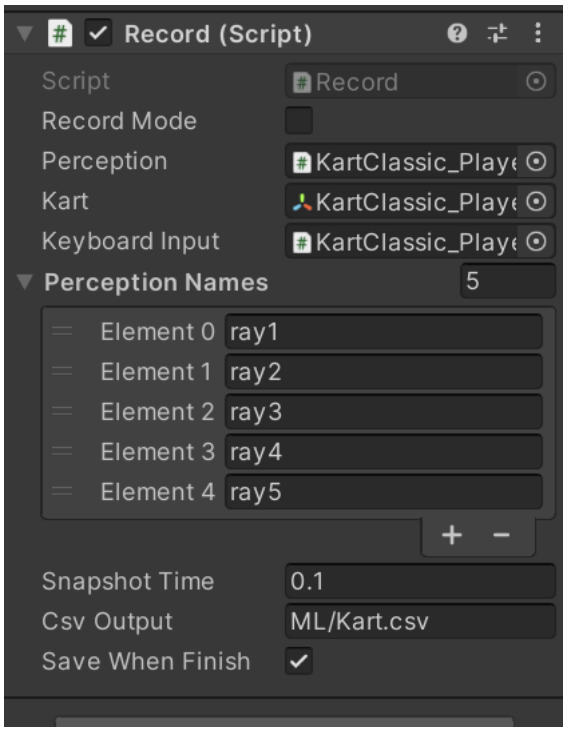
```
RIGHT_ACCELERATE=4, LEFT_BRAKE=5, RIGHT_BRAKE=6 }
```

En la implementación actual, cuando un rayo no detecta nada, alamcena el valor de -1 para indicarlo.

La idea es jugar varias veces con el record activado y almacenar las trazas. Para ello basta con cargar MainScene y darle al play.




Con el RecordMode de Record activo (ten cuidado no machacar los ficheros de partidas previas, cambia el nombre en cada ejecución)




Snapshot Time es el tiempo en segundos que pasa entre cada captura de datos. Puedes jugar ocn el para generar más datos o menos e nfunción de tus necesidades

## Ejercicio 2 visualiza los datos (1 punto):

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

Ahi dibuja la distribución de clases que hayais grabado. Ten cuidado porque

tendrá seguramente más de 3 atributos y no podréis visualizarlo directamente.

## Ejercicio 3 Limpia el dataset (1 punto):

Crea una versión dle 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 corrigelos.

Tambien en esta sección debes normalizar los datos.

## Ejercicio 4 Prueba diferentes modelos de Machine Learning (hasta 6 puntos):

- Al menos uno de ellos debe ser vuestra implementación del perceptrón multicapa. El perceptrón multicapa debe permitir crear modelos de cualquier numero de capas. Al menos en el jupiter notebook a entregar debe haber un ejemplo con más de 3 capas.
- 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 logistica o sigmoidal para toda la red)
- A parte de esta versión, podeis cacharear con diferentes parametros de la versión de SKLearn del MLPClassifier (con función de activación relu o con diferentes versiones del optimizador) **(3 puntos)**
- Elige al menos otros dos modelos diferentes al MLP para comparar los resultados. Puedes elegir más de 2 y pueden ser todo lo ocmplejos que quieras (Incluido Deep Learning) cuanto mas experimentéis con esta parte y mas complejos sean los modelos más nota tendréis. **(hasta 2 puntos en función de los modelos implementados)**
- Debes dividir los datos entre validación y test y si es necesario usar cross_validation.
- Muestra las matrices de confusión de todos los modelos usados, también calcula accuracy y MSE para todos los modelos.
- Al final en el propio Notebook usando Markdown explica que modelo crees que se adapta mejor al juego y cual elegirías. **(1 punto)**
- Itera entre los modelos, sus hiperparámetros, los datos exportados y la limpieza de los mismos, asi como el numero 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 fontanerái necesaria para poder implementar el modelo de MLPClassifier en Unity. Para poder exportar dicho modelo a diferentes formatos podéis usar el fichero 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.

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

```
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 numero de parameter---
```

- num_layer: nos indica le numero 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 codigo introduce los sesgos dentro de las matrices, aunque tendréis que cambiar codigo en Unity. También podéis sacar en la exportación los sesgos para poder pasar la info a 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.

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

```
public float[] FeedForward(Perception p, Transform transform)
{
    Parameters parameters = Record.ReadParameters(8, Time.
    timeSinceLevelLoad, p, transform);
    float[] input=parameters.ConvertToFloatArrat();
    Debug.Log("input " + input.Length);

    //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
        //has assumed that they do not exist.
        return new float[7];
    }
```

Y la función ConvertIndexToLabel ya que según como juegues puede que no uses todas las clases y por tanto el orden de las mismas en el MLP será diferente.

```
    public Labels ConvertIndexToLabel(int index)
    {
        //TODO: implement the conversion from index to actions.
        return Labels.NONE;
    }
```

Puedes necesitar tocar algo de código adicional para implementar el MLP aunque hemos intentado que sea el menor posible. En cualquier caso, siéntete libre de cambiar lo que necesites.

Lo ideal es conseguir un modelo que permita al Kart llegar a la meta y que sea razonablemente similar a nuestra forma de jugar, pero **si no se consigue no pasa nada debido a las limitaciones de tiempo que disponemos.**

### Ejercicio 6 Implementa otro ejecutar de un modelo de ML del os que hayas usado en Unity (opcional + 1 punto)

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 puntuacuón se sumará a la puntación del examen.

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

## English Version

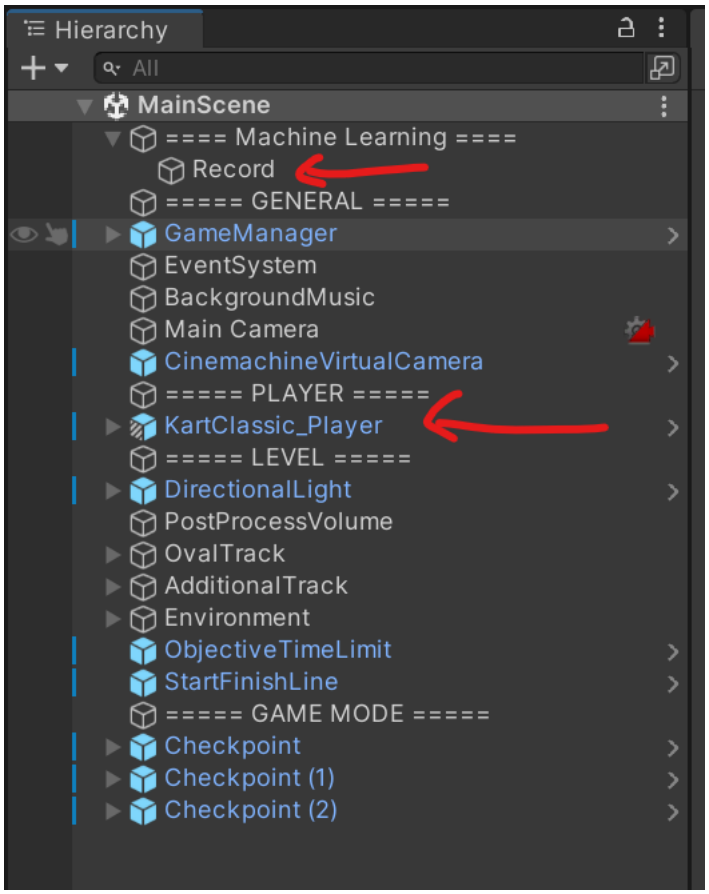
We belong to a videogame studio that intends to develop a karting game similar to Super Mario Kart. We have developed a playable prototype that you have at your disposal in the virtual campus.

Your task is to develop an AI for the cars controlled by the game. To do this, we will train an agent by playing the game multiple times. Once we have saved the example data, our task is to perform a theoretical study on which Machine Learning model is more efficient to implement the agent.

### Exercise 1 Play the game several times:

In the attached Unity project, loading MainScene we can see the following hierarchy:

The GameObject KartClassic_Player has the car control implementation. Two components have been added, Perception and MLAgent:




- Perception: is a class that implements the perception of the agent. You can modify it as much as you want to add more information to the game state. But by default it can generate a number of physical rays indicated in the Rays field of a distance indicated in Distance. The layer it hits is layer 11, which is the track layer. There is a field, HeightOffest that allows us to raise a little the ray so that it does not hit against the road but against the edges of the same one.
- MLAgent: allows us to implement a Machine Learning model. It is prepared to implement several models, but by default it only partially implements the MLPClassifier of SkLearn. When the AgentEnable check is active, the

player will read the imput from the MLAgent and not from the input. In text we have the asset with the serialized MLPClassifier model previously generated from Python.


The GameObject Record allows us to record a game. If we have the RecordMode check active when starting a game, the execution is saved. The file name is indicated in Csv Output and is automatically saved if you have the Save When Finish check active. You can also start the recording and end it by pressing the R key.

Perception Names is the name we give to the ray fields. In addition to these we save: - kartx: x position of the Kart. - karty: position y of the Kart. - kartz: position z of the Kart. - time: time in seconds where the data capture is taken.

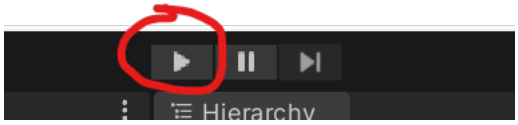
- action: action performed by the player.

The possible actions are:

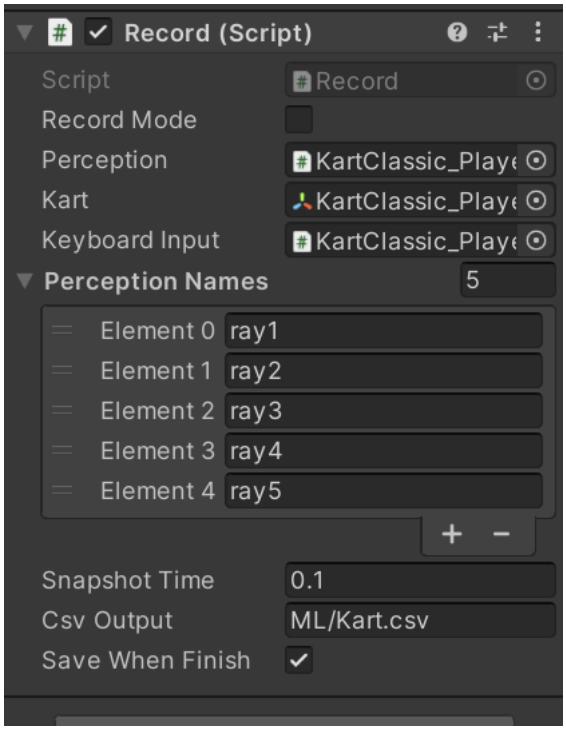
```
public enum Labels { NONE=0, ACCELERATE=1, BRAKE=2, LEFT_ACCELERATE=3,
 RIGHT_ACCELERATE=4, LEFT_BRAKE=5, RIGHT_BRAKE=6 }
```

In the current implementation, when a ray does not detect anything, it stores a value of -1 to indicate this.

The idea is to play several times with the record activated and store the traces. To do this, just load MainScene and hit play.




With the RecordMode of Record active (be careful not to crush the files of previous games, it changes the name in each execution)




## Exercise 2 visualize the data (1 point):

Create a CSV with all the game traces you have generated and take it to Python / Jupiter Notebook.

There draw the distribution of classes that you have recorded. Be careful because it will probably have more than 3 attributes and you will not be able to view it directly.

## Exercise 3 Clean the dataset (1 point):

Create a clean version of the dataset with the relevant information both in attributes used and actions. Check that there are no erroneous or meaningless values and correct them.

Also in this section you should normalize the data.

## Exercise 4 Try different Machine Learning models (up to 6 points):

- At least one of them must be your implementation of the multilayer perceptron. The multilayer perceptron must allow you to create models with any number of layers. At least in the jupiter notebook to be delivered there must be an example with more than 3 layers.
- Check that the results of your implementation are similar (but not identical) to the MLPClassifier of SKLearn (set the same values of alpha, learning_rate_init and use the logistic or sigmoidal function for the whole network).
- Apart from this version, you can cache with different parameters of the SKLearn version of the MLPClassifier (with relu activation function or with different versions of the optimizer) **(3 points)**.
- Choose at least two other models different from the MLP to compare the results. You can choose more than 2 and they can be as complex as you want (including Deep Learning) the more you experiment with this part and the more complex the models are the more points you will get. (up to 2 points depending on the models implemented) **(up to 2 points depending on the models implemented)**
- You should split the data between validation and test and if necessary use cross_validation.
- It shows the confusion matrices of all the models used, it also calculates accuracy and MSE for all the models.
- At the end in the Notebook itself using Markdown explain which model you think best fits the game and which one you would choose. **(1 point)**
- Iterate between the models, their hyperparameters, the exported data and their cleanliness, as well as the number of training examples to get models with the best possible theoretical performance.

### Exercise 5 Implement the multilayer perceptron of your choice in Unity (2 points).

You can choose your model or the MLPClassifier model from SKLearn. By default comes the fontanerai needed to implement the MLPClassifier model in Unity. To export the model to different formats you can use the file Utils.ExportAllformatsMLPSKlearn that exports the model to different formats: pickle, onnx, JSON and a custom format that allows you to export your model if you consider it.

The custom format behaves as follows (each field is a line):

```
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]
--- Repeat for all layers / theta, change parameter number---
```

- num_layer: it indicates the number of layers of the perceptron.
- dims: is the dimension of the matrix
- name: is the name of the coefficient. The theta matrices are called coefficient and the biases are called intercepts. This is the default format, but you can change it if your code introduces the biases inside the matrices, although you will have to change the code in Unity. You can also take out the biases in the export to be able to pass the info to Unity.

The current implementation reads correctly the MLPClassifier parameters from SKLearn. To get the points of this exercise you should at least use that implementation, but it will be valued to use your own implementation in the agent.

To implement the agent you have to write the following functions in C# in the MLAgent component

```
public float[] FeedForward(Perception p, Transform transform)
{
    Parameters parameters = Record.ReadParameters(8, Time.
    timeSinceLevelLoad, p, transform);
    float[] input=parameters.ConvertToFloatArrat();
    Debug.Log("input " + input.Length);

    //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
        //has assumed that they do not exist.
        return new float[7];
    }
```

And the ConvertIndexToLabel function because depending on how you play you may not use all the classes and therefore the order of the classes in the MLP will be different.

```
    public Labels ConvertIndexToLabel(int index)
    {
        //TODO: implement the conversion from index to actions.
        return Labels.NONE;
    }
```

You may need to touch some additional code to implement the MLP although we have tried to keep it as little as possible. In any case, feel free to change whatever you need.

The ideal is to get a model that allows the Kart to reach the goal and that is reasonably similar to the way we play, but **if this is not achieved it's ok due to the time constraints we have.**

## Exercise 6 Implement another run of an ML model of the ones you have used in Unity (optional + 1 point)

Performing this section adds **one extra point** to the practice grade. If the practice itself has a 10 or a 9 something, the rest of the score will be added to the exam score.

You can choose any other model runner you want, Decision Tree, KNN or more complex models with deep learning. If you are going to use more complex models we recommend you to use ML-Agents but if you want to implement something simple, you can do it with models like KNN or decision tree where creating an executor is more or less easy to do.

If you do this section, explain in the Notebook if it has worked better in practice than the MLP model you have implemented in the previous exercise.