# Creación de agentes inteligentes para videojuegos con Deep Q-Learning utilizando Unity y ML-Agents.

---

## 1. Introducción

Aprender interactuando con el entorno es probablemente lo primero que se nos viene a la mente cuando pensamos acerca de la naturaleza del aprendizaje. Por ejemplo, los niños tienden a tener comportamientos erráticos que van desapareciendo a medida que crecen, pero son por medio de estos que los niños interactuan con el entorno, aprenden que comportamiento puede ser beneficioso, acerca de las consecuencias de las acciones y que deben hacer para conseguir ciertas metas. Este paradigma en *Machine Learning* es conocido como aprendizaje por refuerzo (*Reinforcement Learning* o *RL*) y es el tema central de este trabajo, en el que utilizaremos un motor de videojuegos para crear una IA que aprenda por medio de algoritmos de *Reinforcement Learning* a moverse en un circuito de carreras.

### ¿Qué es Reinforcement Learning? y ¿Cómo aplicarlo?

*Reinforcement Learning* es aprender qué se debe hacer interactuando con el entorno para maximizar una recompensa numérica. El sujeto que quiere maximizar la recompensa no sabe que acciones debe tomar pero sabe que acciones puede tomar y debe descubrir cuáles acciones tienen como consecuencia una mayor recompensa por medio de prueba y error. También es importante reconocer que no todas las acciones solo tienen una recompensa inmediata, algunas también tienen recompensas a largo plazo.

Para poder resolver problemas de *Rinforcement Learning* vamos a ver el concepto de *Markov Decision Processes* o *MDP*. En un MDP tenemos a un tomador de decisiones al que llamaremos agente (*agent*), este interactúa con el entorno en el que se encuentra (*environment* o *env*). En cada paso (*step*), el agente obtiene una representación del estado del entorno (*state*). Con esta representación, el agente tomará alguna acción (*action*). El entorno entonces cambiará a un nuevo estado y el agente adquiere una recompensa como consecuencia de sus acciones previas. La ciclo de recibir el estado del entorno, tomar una acció y obtener una recompensa es también llamado trayectoria (*trajectory*). Si esta explicación de lo que son los MDP parece familiar, es porque es una perfecta descripción de lo que queremos hacer al usar *Reinforcement Learning*.

Componentes de un *MDP*:

- Agente.
- Entorno.
- Estado.
- Recompensa.

Notación de un MDP:
En un MDP tenemos un conjunto de estados $S$, un conjunto de acciones $A$, y un conjunto de recompensas $R$. Se asume que todos los conjuntos son finitos.

En cada paso de tiempo $t = 0, 1, 2, \cdots$, el agente recibe una representación el estado del entorno $S_t \in S$. Basado en este estado, el agente selecciona una acción $A_t \in A$. Esto nos da un pareja de estado-

acción $(S_t, A_t)$.

Entonces el tiempo incrementa al siguiente paso de tiempo $t+1$ y el entorno cambia al nuevo estado $S_{t+1} \in S$. Para este tiempo, el agente recibe una recompensa $R_{t+1} \in R$ para la acción $A_t$ tomada en el estado $S_t$.

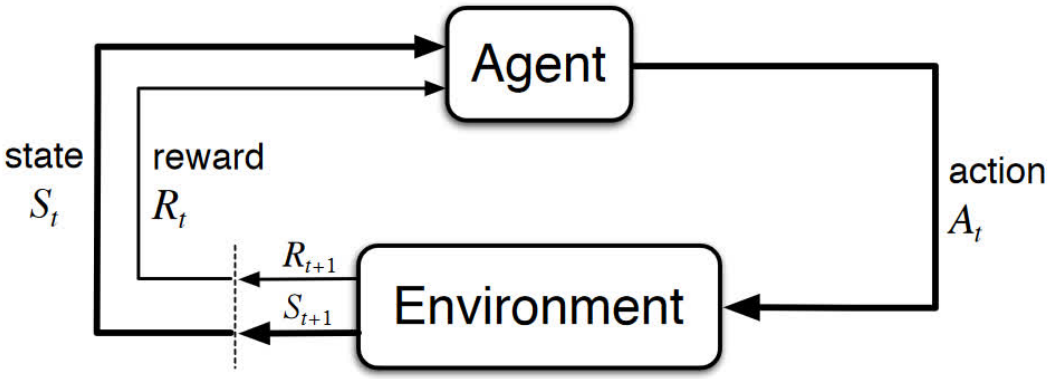
Podemos pensar en este proceso de recibir una recompensa como una función arbitraría $f$ que mapea las parejas estado-acción a recompensas. Entonces para cada tiempo $t$, tenemos:

$$f(S_t, A_t) = R_{t+1}$$

También podemos representar la trayectoria como:

$$S_0, A_0, R_1, S_1, A_1, R_2, \cdots$$

Este diagrama muestra claramente la idea detrás de un MDP:




Vamos a describir el proceso que se muestra en el diagrama:

1. Para el tiempo $t$ el entorno se encuentra en el estado $S_t$.
2. El agente observa el estado del entorno y selecciona una acción $A_t$.
3. El entorno cambia al estado $S_{t+1}$ y el agente adquiere una recompensa $R_{t+1}$
4. Este proceso se repite de nuevo para un paso en tiempo $t+1$.
    - Nota: $t+1$ ya no es el paso en el siguiente tiempo, ahora es el presente. Cuando cruzamos la linea punteada abajo a la izquierda, el diagrama muestra como $t+1$ se transforma en el paso actual $t$, entonces $S_{t+1}$ y $R_{t+1}$ ahora se muestran como $S_t$ y $R_t$ respectivamente.

### ¿Qué es Deep Q-Learning?

Para hablar de *Deep Q-Learning* primero debemos saber un poco sobre *Q-Learning*. *Q-Learning* es un algoritmo de *Reinforcement Learning* para que busca optimizar una función $q_*$ para la toma de decisiones similar al ejemplo de un niño aprendiendo a interactuar con el entorno, al inicio el algoritmo opta por la exploración pero a medida que avanza y aprende sobre el entorno y las acciones que excoge comienza a ser más "codicioso" y tiende a explorar menos, intentando máximizar la recompensas obtenidas con todo lo previamente aprendido sin arriesgarse tanto al explorar.

Sin embargo, esta técnica no suele adaptarse bien a entornos complejos, pero conocemos una herramienta que si lo hace, las redes neuronales. De ahí su nombre *Deep Q-Learning*, donde utilizaremos redes neuronales para aproximar los valores de la función $q_*$.

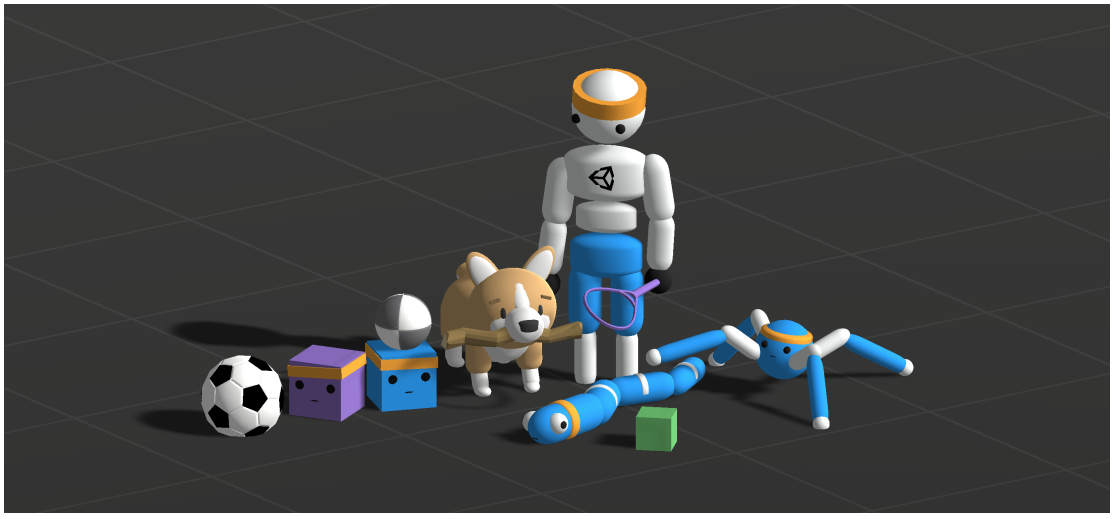
Hablaremos más sobre esto en la parte del entrenamiento.

### ¿Que es Unity?

*Unity* fue diseñado como un software para la creación de videojuegos, pero con el tiempo se ha convertido en una plataforma para crear contenido interactivo. Unity tiene un motor de físicas y renderizado robustos así como una interfaz gráfica llamada Unity Editor.

La plataforma de Unity ya ha sido adoptado en industrias como el Gaming, Arquitectura, Ingeniería, Construcción, Automotriz y Cinematográfica. Es usada por gran parte de la comunidad de desarrolladores de videojuegos para crear una gran variedad de proyectos interactivos de simulación que van desde un pequeño dispositivo mobil, hasta consolas de videojuegos y experiencias AR/VR.

## ¿Que es ML-Agents?



El *Unity Machine Learning Agents Toolkit* o *ML-Agents* es un proyecto open-source que le permite a desarrolladores e investigadores simular diferentes entornos en el Unity Editor e interactuar con ellos utilizando un API de Python.

Algunas de las características más importantes de ML-Agents son:

- Soporte para multiples configuraciones y escenarios de entornos (*environments*).
- Soporte para entrenamiento de un solo agente o multiples agentes, ya sea en escenarios cooperativos o competitivos.
- Soporte para diferentes algoritmos de *Deep Reinforcement Learning* (PPO, SAC, MA-POCA).
- Soporte para dos algoritmos de *Imitation Learning* (BC y GAIL).
- Permite una definición sencilla de escenarios complejos para *Curriculum Learning*.
- Entrenamiento de agentes con entornos cambiantes.
- Utiliza el *Unity Inference Engine* para proveer soporte multiplataforma de forma nativa.
- Control de los entornos con Python.
- La posibilidad de mover tus entornos de Unity a gym.

## ¿Qué vamos a hacer?

Como el título sugiere, vamos a utilizar Unity y ML-Agents para crear una IA de videojuegos, inspirados en los juegos de carreras 3D, decidimos que el objetivo de este proyecto es crear una IA que pueda manejar por un circuito de carreras sin ser explicitamente programada para ello, solo se programaran sus acciones y recompensas. Por ejemplo, seguir en el camino aumentará las recompensas obtenidas, mientras que salir del circuito terminará inmediatamente con la simulación, pero es la IA quien tendrá que descubrir esto.

Entraremos en más detalles en las siguientes secciones.

### Tecnologías utilizadas

| Tecnología | Versión |
|---|---|
| Unity | 2019.3.1f1 |
| ML-Agents (Plugin) | 1.9.1 (Preview) |
| ml-agents (python) | 0.25.1 |
| ml-agents-envs | 0.25.1 |
| Python | 3.9 |
| Pytorch | 1.8.1 + CPU |

### Equipo

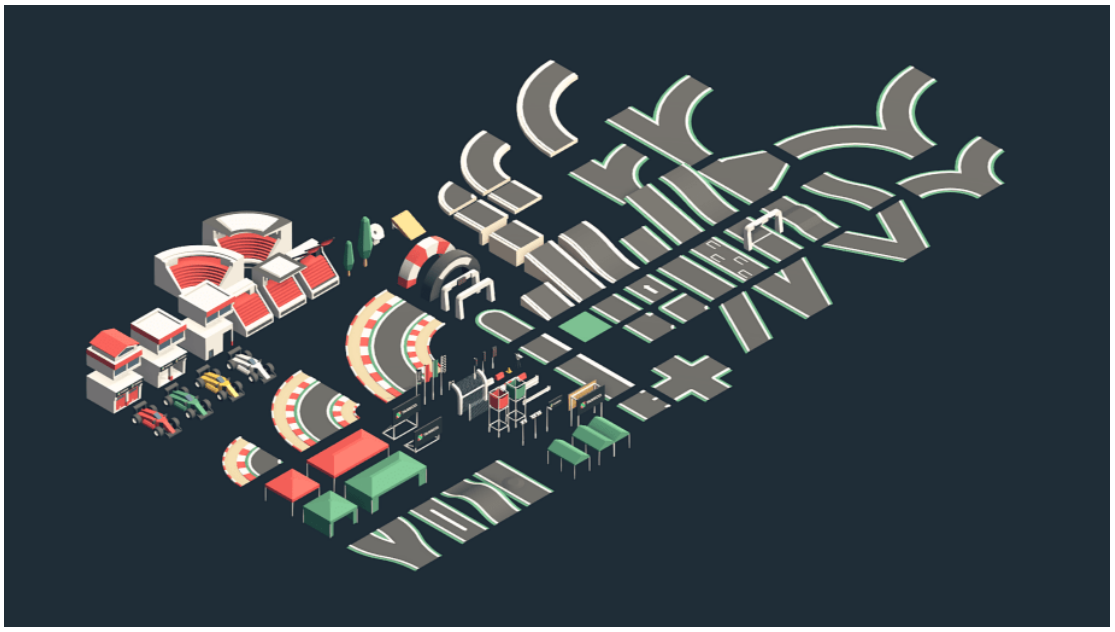
1. Aguilera Luzania José Luis.
2. Baez Camacho Jesús Armando.
3. Castro Marquez Francisco Javier.

---

## 2. Primeros pasos con Unity

En esta sección se analizará la estructura del proyecto desde la vista de Unity.

### Assets utilizados.



Para la simulación del entorno usaremos los assets de la página de Kenney, en específico el Racing Kit

### Paquetes del proyecto.

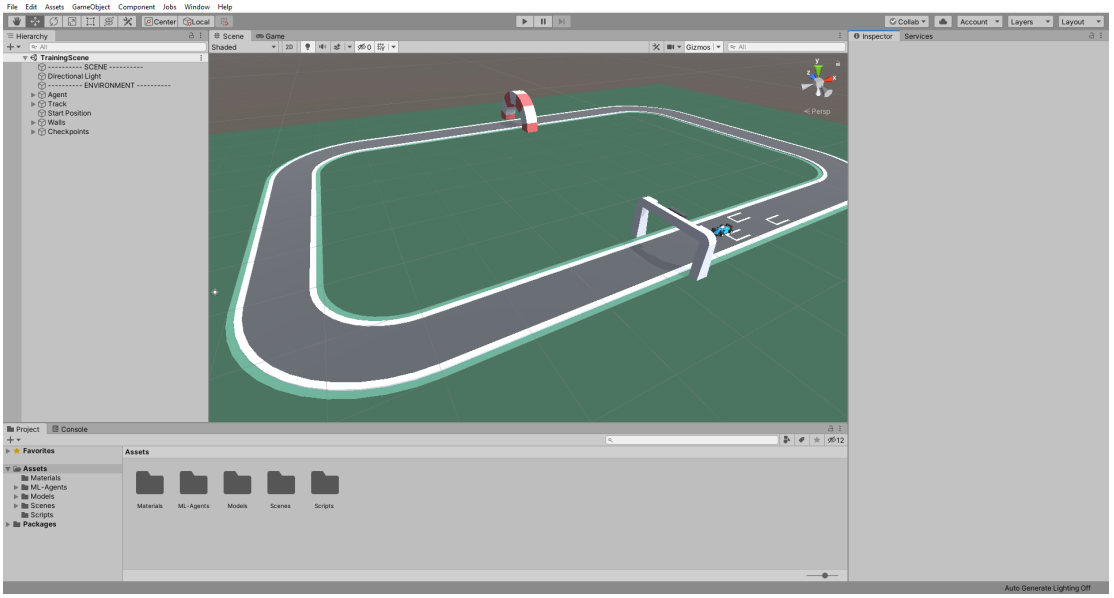
Los paquetes del proyecto pueden ser encontrados a través del Package Manager.

| Paquete | Versión |
|---|---|
| ML Agents | 1.9.1 (Preview) |
| Rider editor | 1.1.4 |
| Test Framework | 1.1.11 |
| TextMesh Pro | 2.0.1 |
| Timeline | 1.2.10 |
| Unity Collaborate | 1.2.16 |
| Unity UI | 1.0.0 |

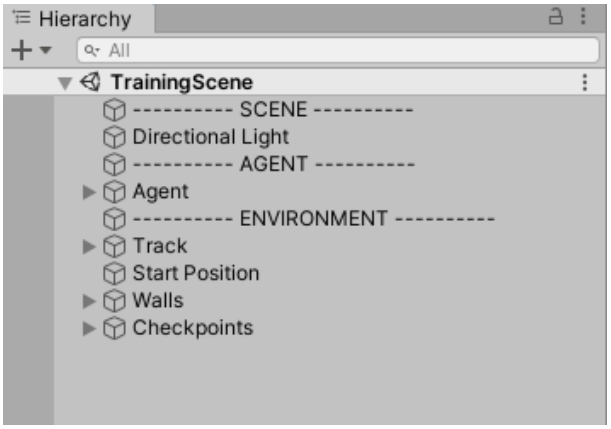
### Estructura del proyecto en Unity.

Al momento de abrir el proyecto en Unity debemos ir a la ventana del proyecto y buscar entre los assets la carpeta que se llama Scenes y abrir hacer doble clic en el archivo que se llama TrainingScene. Esto cargará una nueva escena en Unity.


El editor debería verse así



En la ventana de jerarquía podemos ver la estructura de la escena.



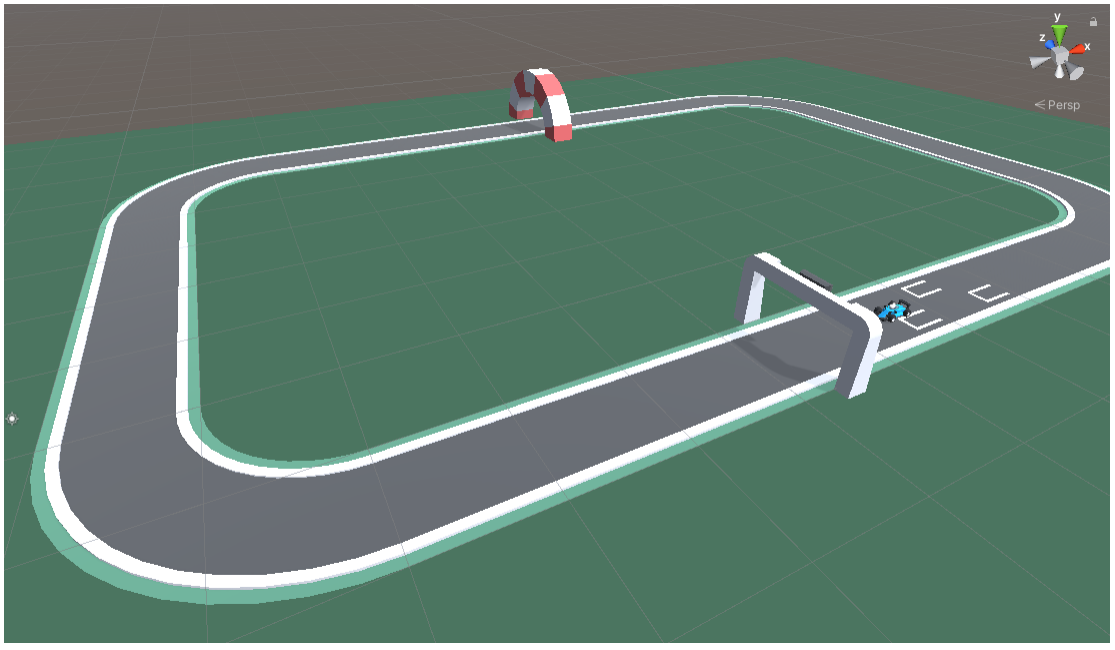

La explicación de la escena es la siguiente:

1. **---------- SCENE ----------**: Es un objeto vacío que se usa como separador.
2. **Directional Light**: Es el objeto encargado de la iluminación.
3. **--------- AGENT ---------**: Es un objeto vacío que sirve como separador.
4. **Agent**: Es el agente.
5. **--------- ENVIRONMENT ---------**: Es un objeto vacío que sirve como separador.
6. **Track**: Es el objeto que contiene todos los modelos que forman el circuito.
7. **Start Position**: Es el objeto que guarda la posición inicial del agente para cada episodio.
8. **Walls**: Es el objeto que contiene todos los muros del circuito.
9. **Checkpoints**: Es el objeto que contiene todos los checkpoints del circuito.

### Estructura del environment.

Durante la explicación del MDP se introdujo el concepto de entorno (*environment*), vamos a ver en que consiste el entorno simulado en Unity.

Como se observa al cargar la escena TrainingScene en Unity el entorno es el siguiente:



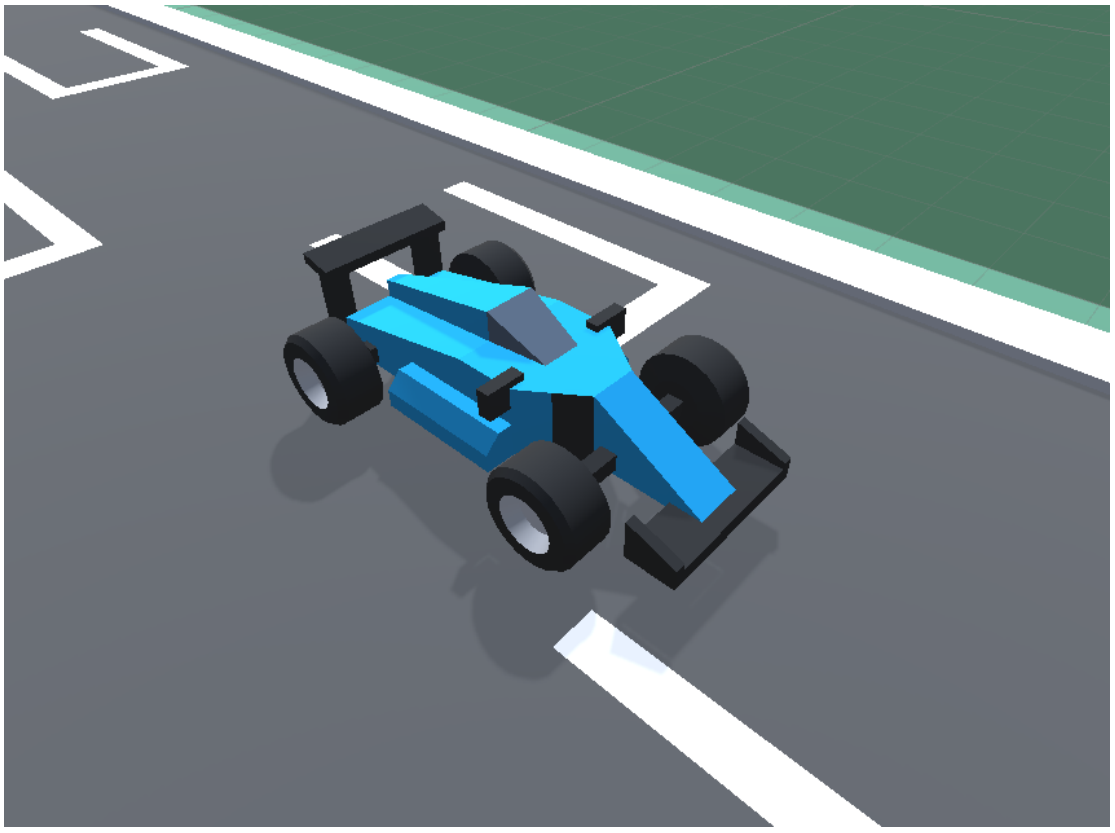

Los componentes del entorno son los siguientes:

1. **Track**: Es un contenedor, contiene todos los objetos 3D que forman el circuito. Los modelos se encuentran en el formato FBX y es el formato que recomendamos usar para cualquier simulación en Unity.

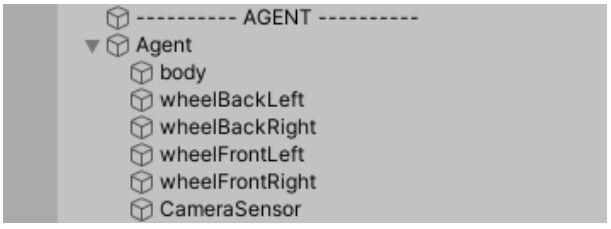
2. **Start Position**: Es un objeto que tiene las coordenadas $x$, $y$ y $z$ que representan la posición inicial del agente en cada episodio.
3. **Walls**: Es un contenedor, contiene todos los límites del entorno (*aunque son invisibles, puden ser vistos al seleccionar el objeto*). Estos límites servirán para terminar los episodios si el agente los toca, es decir, se salio de la pista.
4. **Checkpoints**: Es un contenedor, contiene objetos en forma de muros invisibles que le darán una recompensa al agente cada vez que pase por ellos, estos solo se pueden atravesar una vez antes de dar la vuelta de nuevo y solo se encuentran en la pista, por lo que el agente recibirá más recompensa al mantenerse en la pista.

### Estructura del Agente.

Es hora de ver a nuestro agente, en este caso nuestro agente es un carro de carreras azul.



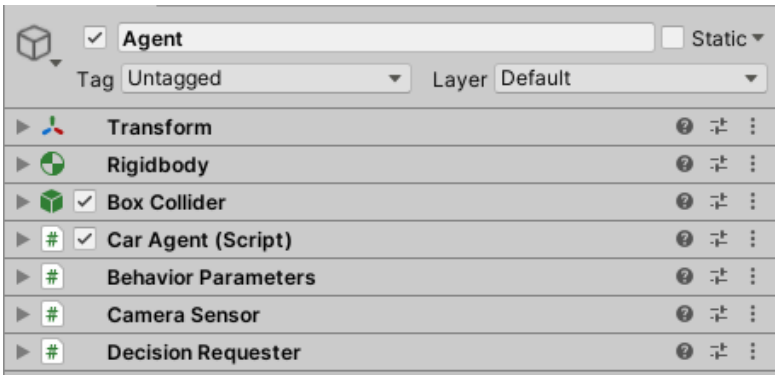
El objeto del agente tiene 6 objetos anidados, los y componentes son:
Los objetos anidados y sus componentes son:




- **body**: Este objeto contiene la información del modelo para la base del carro y sus materiales.
- **wheelBackLeft**: Contiene la información dle modelo de la llanta trasera izquierda.
  - Contine un componente Wheel Collider para la simulación de las físicas en llantas.
- **wheelBackRight**: Contiene la información dle modelo de la llanta trasera derecha.
  - Contine un componente Wheel Collider para la simulación de las físicas en llantas.
- **wheelFrontLeft**: Contiene la información dle modelo de la llanta frontal izquierda.
  - Contine un componente Wheel Collider para la simulación de las físicas en llantas.
- **wheelFrontRight**: Contiene la información dle modelo de la llanta frontal derecha.

    - Contine un componente Wheel Collider para la simulación de las físicas en llantas.
- **Camera Sensor**: Contiene un componente *camera* que simula una cámara de verdad y la imagen que renderiza es la imagen utilizada por el agente.

Los componentes del agente en Unity son:




1. **Transform**: Guarda información sobre la *posición*, *rotación* y *escala* del objeto. Es un componente básico de Unity, todos los objetos creados dentro del *Unity Editor* lo tienen por defecto.
2. **RigidBody**: Componente engargado de registrar el objeto en el motor de físicas y administrarlas.
3. **Box collider**: Componente que se encarga de gestionar las colisiones dentro de un área definida.
4. **Car Agent (Script)**: Componente que define el comportamiento de las acciones que puede seleccionar el agente. Deriba de la clase Agent de ML-Agents.
5. **Behavior Parameters**: Componente de ML-Agents que permite definir el nombre del comportamiento, el número y tipo de observaciones, el número y tipo de acciones. Además este componente es el encargado de registrar al agente dentro de Unity Environment para la comunicación con Python.
6. **Camera Sensor**: Componente que simula una cámara, también se encarga de recoger observaciones (4 imagenes PNG en blanco y negro de 84×84 pixeles) y enviarlas al Unity Environment.
7. **Decision Requester**: Componente que se encarga de solicitar decisiones por parte del Agente cada cierto tiempo. No es necesario y puede llamarse mediante código.

Vamos a ver el código del agente para confirmar que el agente no fue programado para navegar por el circuito.

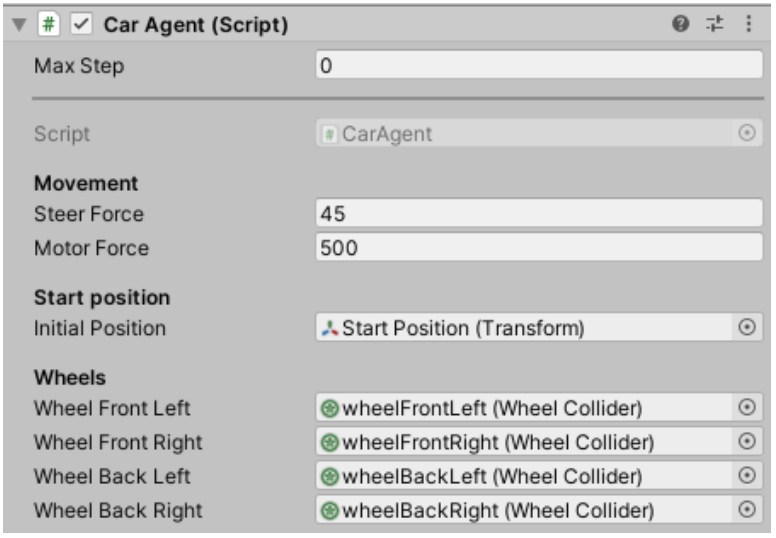
**CarAgent (Script)**:

```
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;

public class CarAgent : Agent
{
    /// <summary>
    /// Steer force.
    /// </summary>
    [Header("Movement")] [SerializeField] private float steerForce;

    /// <summary>
    /// Motor force.
    /// </summary>
    [SerializeField] private float motorForce;
```

```
    /// <summary>
    /// The starting position where the agent will start in each episode.
    /// </summary>
    [Header("Start position")] [SerializeField] private Transform initialPosition;

    /// <summary>
    /// WheelCollider component for the Wheel Front Left.
    /// </summary>
    [Header("Wheels")] [SerializeField] private WheelCollider wheelFrontLeft;

    /// <summary>
    /// WheelCollider component for the Wheel Front Right.
    /// </summary>
    [SerializeField] private WheelCollider wheelFrontRight;

    /// <summary>
    /// WheelCollider component for the Wheel Back Left.
    /// </summary>
    [SerializeField] private WheelCollider wheelBackLeft;

    /// <summary>
    /// WheelCollider component for the Wheel Back Right.
    /// </summary>
    [SerializeField] private WheelCollider wheelBackRight;

    /// <summary>
    /// Reference to this transform component.
    /// </summary>
    private Transform _transform;

    /// <summary>
    /// Function that is called when starting a new episode and
    /// is responsible for restarting the agent.
    /// </summary>
    public override void OnEpisodeBegin()
    {
        _transform = transform;
        _transform.position = initialPosition.position;
        _transform.eulerAngles = new Vector3(0f, 90f, 0f);
    }

    /// <summary>
    /// Function that is in charge of determining the behavior depending on the actions it
receives.
    /// Actions: 0: go straight | 1: Steer to the right | 2: Steer to the left.
    /// </summary>
    /// <param name="actions"></param>
    public override void OnActionReceived(ActionBuffers actions)
    {
        int action = actions.DiscreteActions[0];
```

```
        float h, v = -motorForce;

        if (action <= 1) h = action;
        else h = -1;

        wheelBackLeft.motorTorque = v;
        wheelBackRight.motorTorque = v;
        wheelFrontLeft.steerAngle = h * steerForce;
        wheelFrontRight.steerAngle = h * steerForce;
    }

    /// <summary>
    /// Function that is invoked when the physics engine detects a collision
    /// between two or more objects.
    /// </summary>
    /// <param name="other">An object that contains all the information about the collision.
</param>
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Target"))
            SetReward(+1f);

        if (other.gameObject.CompareTag("Wall"))
        {
            SetReward(-5f);
            EndEpisode();
        }
    }
}
```

este código dentro del Unity Editor se ve de la siguiente manera:




como podemos observar, desde el Unity Editor podemos asignarle valores a las variables declaradas en el script. Para este agente no nos interesa tener un número máximo de pasos por el momento, la fuerza de giro para las llantas es de 45u y la fuerza del motor es de 500u. Utilizamos el objeto Start Position para obtener la posición inicial y hacemos referencia a los "Wheel Collider" de todas las llantas.

Para entender mejor al agente debemos saber que tiene tres acciones de tipo discretas, debido a que solo puede escoger una de tres acciones [0: No hacer nada | 1: Girar a la derecha | 2: Girar a la izquierda].

Para el agente, las funciones que realmente nos interesan son la función OnActionReceived y OnTriggerEnter .

1. La función OnActionReceived se encarga de verificar que tipo de acción recibio el agente y que debe hacer según la acción, si recibio 0 el agente continua avanzando, si recibio un 1 el agente gira a la derecha, si recibio un 2 el agente gira a la izquierda.
2. La función OnTriggerEnter es invocada cuando el motor de físicas detecta una colisión entre dos objetos, en este caso si el agente colisiona con un objeto que tenga la etiqueta "Target" obtendrá una recompensa de +1.0. Por otro lado si el agente colisiona con un objeto que tenga la etiqueta "Wall" obtendrá una recompensa de -5.0 y terminara el episodio.

Ahora ya entendemos mejor como esta estructurado el proyecto en Unity, desde el entorno hasta nuestro agente. Es hora de ver como combinar Unity y Python para darle vida a nuestro agente!

---

# 3. Dependencias de Python

Utilizaremos las siguientes dependencias:

- matplotlib: para mostrar los resultados y otros gráficos.
- numpy: para operaciones numéricas.
- torch: para la aplicación de redes neuronales.
- random: para la aleatorización.
- ml_agents_envs: para la comunicación con Unity.
- typing: para un manejo sencillo de elemntos como diccionarios, tuplas, listas, etc.

```
import matplotlib.pyplot as plt
import numpy as np
import sys
import torch
import random
from mlagents_envs.environment import UnityEnvironment
from mlagents_envs.environment import ActionTuple, BaseEnv
from typing import Tuple
from typing import NamedTuple, List
from typing import Dict
from math import floor
```

---

# 4. Primeros pasos con ML-Agents

Antes de comenzar a definir nuestra red neuronal, entrenar a nuestro agente y ganar algún premio por hacer algo totalmente innovador, vamos a repasar un lo básico de la API de Python de ML-Agents.

### ¿Cómo conectar ML-Agents con Unity a través de Python?

Para conectar Unity y Python debemos usar la función `UnityEnvironment()`, la función crea un nuevo entorno y establece la conexión con Unity, una vez que se ejecuta la función, dentro del Unity Editor se debe presionar el botón "Play" para completar la conexción. Cada vez que la función `UnityEnvironment()` es invocada, también se debe invocar la función `close()` cuando se quiera cerrar el entorno y desconectar Unity.

El código para conectar nuestro entorno (que llamaremos env) es el siguiente:

```
env = UnityEnvironment(file_name = None, base_port=5004)
env.reset() # <--- Reinicia la simulación.
```

### ¿Cómo obtener información de los Agentes?

Si queremos obtener información de los agentes debemos usar los atributos del entorno, por ejemplo:

- Para obtener el nombre del comportamiento del agente usamos el atributo `behavior_specs` que regresa un mapeo de los nombres de comprtamiento a las tuplas de observaciones y acciones. Convertimos en una lista y obtenemos el primer elemento para conocer el nombre, que en el caso de este proyecto es "CarAI".
- Para obtener las específicaciones del comportamiento usamos el atributo `behavior_specs`, solo que ahora buscamos las específicaciones con el nombre del comportamiento y usamos el atributo `observation_specs` para ver como son las observaciones del agente. Para este caso el agente tiene observaciones con la forma (84, 84, 4) que hacen referencia a las 4 imagenes de 84×84 pixeles.
- Para conocer el número de observaciones solo calculamos el número de elemntos que hay en `observation_specs`.
- Como nuestro agente obtiene observaciones visuales, revisemos usando el atributo `shape` para ver si existe una observación con tres parámetros, si existe lo guardamos en una variable e imprimimos {verdadero o falso}.
- Para obtener el tipo de acciones del agente usaremos los atributos `continuous_size` y `discrete_size` para saber el número de ramas de acción de cada tipo. En el caso de nuestro agente, solo tiene una rama de acciones discretas.

```
# Nombre del primer comportamiento.
behavior_name = list(env.behavior_specs)[0]
print(f"Behavior name: {behavior_name} \n")

# Específicaciones del comportamiento.
spec = env.behavior_specs[behavior_name]
print(spec.observation_specs)

# Número de observaciones.
print(f"\nNúmer of observations: {len(spec.observation_specs)}")

# ¿Hay una observación visual?
vis_obs = any(len(spec.shape) == 3 for spec in spec.observation_specs)
print(f"\nVisual observation: {vis_obs}")

# ¿La acción es continua o multi discreta?
if spec.action_spec.continuous_size > 0:
    print(f"\nThere are {spec.action_spec.continuous_size} continuous actions.")
```

```
if spec.action_spec.is_discrete():
    print(f"\nThere are {spec.action_spec.discrete_size} discrete actions.")
```

### La primera observación visual del Agente.

Con el fin de poder ver la primer observación visual de nuestro agente, tenemos que avanzar un paso en el tiempo con el método `step()` del entorno. Pero antes de poder hacerlo debemos revisar los `decision_steps` y los `termina_steps` que se obtienen del método `get_steps(behavior_name)`.

- `decision_steps`: Es una NamedTuple que contiene observaciones, recompensas, el Id del agente y las acciones del agente.
- `terminal_steps`: Es una NamedTuple que contiene observaciones, recompensas, el ide del agente y las banderas de interrupción que el agente tuvo en el episodio terminado el último paso.

Después establecemos las acciones de los agentes para el siguiente paso usando el método `set_action()` que recibe el nombre del comportamiento de los agentes y una ActionTuple de acciones continuas y/o discretas. En este caso no hay acciones que tomar así que se enviará un ActionTuple vacia. Finalmente usamos el método `step()`.

```
decision_steps, terminal_steps = env.get_steps(behavior_name)
env.set_actions(behavior_name, spec.action_spec.empty_action(len(decision_steps)))
env.step()
```

Ahora usamos matplotlib para reconstruir la imagen obtenida de las observaciones y mostrarla.

```
%matplotlib inline

for index, obs_spec in enumerate(spec.observation_specs):
  if len(obs_spec.shape) == 3:
    print("Here is the first visual observation.")
    print(decision_steps.obs[index][0,:,:,:].shape)
    plt.imshow(decision_steps.obs[index][0,:,:,:])
    plt.show()

for index, obs_spec in enumerate(spec.observation_specs):
  if len(obs_spec.shape) == 1:
    print(f"First vector observations: {decision_steps.obs[index][0,:]}")
```

### Probar el seguimiento de un Agente.

Para probar que todo esta funcionando, usamos este bloque de código dar segumiento a nuestro agente.
Se harán 3 episodios, para entender mejor lo que esta sucediendo en este bloque es recomendable volver a leer la introducción a los MDP.

```
for episode in range(3):
    env.reset() # <--- Reiniciamos el entorno.
    decision_steps, terminal_steps = env.get_steps(behavior_name)

    # -1 Sin seguimiento.
```

```
    tracked_agent = -1

    # Para el agente en segimiento.
    done = False
    episode_rewards = 0

    while not done:
        # Seguimiento del primer agente si este no esta en seguimiento.
        # Nota: len(decision_steps) = [number of agents that requested]
        if tracked_agent == -1 and len(decision_steps) >= 1:
            tracked_agent = decision_steps.agent_id[0]

        # Generar las acciones para todos los agentes.
        action = spec.action_spec.random_action(len(decision_steps))

        # Establecer las acciones.
        env.set_actions(behavior_name, action)

        # Movemos la simulación al siguiente step.
        env.step()

        #Recopilar los resultados de la simulación.
        decision_steps, terminal_steps = env.get_steps(behavior_name)

        # El agente solicitó una decisión.
        if tracked_agent in decision_steps:
            episode_rewards += decision_steps[tracked_agent].reward
        if tracked_agent in terminal_steps:
            episode_rewards += terminal_steps[tracked_agent].reward
            done = True

    # Se imprimen los resultados del episodio.
    print(f"Total rewards for episode {episode} is {episode_rewards}.")

# Se cierra en environment.
env.close()
print("Closed environment.")
```

---

# 5. Definiciones de VisualQNetwork, Experiencia y Entrenamiento.

Ya tenemos una idea de la definición de Reinforcement Learning, MDP y como Python se conecta con Unity, ahora vamos a definir las cosas que necesitamos para implementar el Deep Q-Learning.

## Definición de la clase `VisualQNetwork`.

Nuestra red es bastante simple, consta de dos convoluciones, un batch normalization y dos capas densas.

- La entrada que esta definida por `input_shape` de nuestra red es un tensor tridimensional porque lo que queremos es enviar nuestras 4 imagenes de 84×84 pixeles. Aunque las dimensiones pueden variar.
- La salida que esta definida por `output_shape` de nuestra red es unidimensional porque para nuestro problema solo hay una rama de posibilidades discretas que tomar.

```
class VisualQNetwork(torch.nn.Module):
    def __init__(
        self,
        input_shape: Tuple[int, int, int],
        encoding_size: int,
        output_size: int
    ):
        """
        Crea una red neuronal que toma como input un batch de imagenes
        (tensor tridimensional) y da como salida un batch de outputs
        (tensor unidimensional).
        """
        super(VisualQNetwork, self).__init__()
        height = input_shape[0]
        width = input_shape[1]
        initial_channels = input_shape[2]
        conv_1_hw = self.conv_output_shape((height, width), 8, 4)
        conv_2_hw = self.conv_output_shape(conv_1_hw, 4, 2)
        self.final_flat = conv_2_hw[0] * conv_2_hw[1] * 32
        self.conv1   = torch.nn.Conv2d(initial_channels, 16, [8, 8], [4, 4])
        self.batchNorm1 = torch.nn.BatchNorm2d(16)
        self.conv2   = torch.nn.Conv2d(16, 32, [4, 4], [2, 2])
        self.dense1 = torch.nn.Linear(self.final_flat, encoding_size)
        self.dense2 = torch.nn.Linear(encoding_size, output_size)

    def forward(self, visual_obs: torch.tensor):
        visual_obs = visual_obs.permute(0, 3, 1, 2)
        conv_1 = torch.relu(self.batchNorm1(self.conv1(visual_obs)))
        conv_2 = torch.relu(self.conv2(conv_1))
        hidden = self.dense1(conv_2.reshape([-1, self.final_flat]))
        hidden = torch.relu(hidden)
        hidden = self.dense2(hidden)
        return hidden

    @staticmethod
    def conv_output_shape(
        h_w: Tuple[int, int],
        kernel_size: int = 1,
        stride: int = 1,
        pad: int = 0,
        dilation: int = 1,
        ):
            """
            Calcula la altura y el ancho de la salida de una convolution layer.
```

```
        """
        h = floor(
          ((h_w[0] + (2 * pad) - (dilation * (kernel_size - 1)) - 1) / stride) + 1
        )
        w = floor(
          ((h_w[1] + (2 * pad) - (dilation * (kernel_size - 1)) - 1) / stride) + 1
        )
        return h, w
```

## Definición de la clase `Experience`.

Ahor avamos a definir la clase `Experience` que no es más que una NamedTuple que contiene los datos de transición de un agente; contiene sus observaciones, las acciones, la recompensa, si este terminó o no, y sus siguientes observaciones.

```
class Experience(NamedTuple):
    """
    Una experiencia contiene los datos de la transición de un agente.
        -Observation
        -Action
        -Reward
        -Done flag
        -Next Observation
    """
    obs: np.ndarray
    action: np.ndarray
    reward: float
    done: bool
    next_obs: np.ndarray

# Una trayectoria es una secuencia ordenada de experiencias.
Trajectory = List[Experience]

# Un búfer es una lista desordenada de experiencias de múltiples trayectorias.
Buffer = List[Experience]
```

## Definición de la clase `Trainer`.

Antes de ver el código y con el fin de entenderlo mejor, veamos un poco sobre las estrategías a utilizar para resolver nuestro problema de Reinforcement Learning.

### Estrategía Epsilon greedy

Primero hablemos de dos términos importantes "exploration" y "explotation", cuando hablamos de "exploration" nos referimos a que nuestro agente escogerá acciones que no ha usado antes y que no sabe si tendrán o no mayor recompensas que las acciones conocidas y cuando hablamos de "explotation" nos referimos a que nuestro agente escogerá acciones que ya sabe que tienen una alta recompensa.

Es natural pensar que el agente siempre debería escoger las acciones que ya conoce y sabe que tendrán mayor recompensa, sin embargo, ¿qué pasa si hay acciones con mayores recompensas?,

nustro agente nunca lo descubrirá porque siempre escoge las acciones conocidas, pero tampoco queremos que siempre este explorando, queremos un balance entre estos dos términos ahí es donde la estrategía epsilon greedy entra en acción.

En esta estrategía definimos una tasa de exploración $\epsilon$ que inicialmente vale 1. Esta tasa de exploración es la probabilidad de que nuestro agente explore el entorno en lugar de explotarlo. Con $\epsilon = 1$ hay un $100\%$ de seguridad que va a explorar.

A medida que el agente aprende del entorno, al iniciar un nuevo episodio $\epsilon$ debería disminuir haciendo que la exploración sea cada vez menos probable, esto permitirá que el agente explore peor a medida que aprende, se centre más en maximizar la recompensa con lo aprendido.

### Q-Learning

Nuestor objetivo es entrenar bajo una táctica (conocida como *Policy*) maximiza el descuento de la recompensa acumulada $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$, donde $R_{t_0}$ es conocido como el *return*. El descuento, $\gamma$, debe ser una constante entre $0$ y $1$ para asegurarse de que la suma converja. Esto hace que la recompensa del futuro lejano tenga mucho menos importancia para nuestor agente que la recompensa a corto plazo.

Cuando hablamos de Q-Learning la idea principal es tener una función $Q^* : State \times Action \rightarrow \mathbb{R}$, que pueda decirnos lo que nuestro *return* debería de ser, si nosotros tomamos una acción en un estado dado, entonces deberiamos construir fácilmente una táctica que maximiza nuestra recompensa:

$$\pi^*(s) = argmax\ Q^*(s, a)$$

Como no tenemos acceso a toda la información del entorno, no tenemos acceso a la función $Q^*$, pero como dejamos en claro al inicio en nuestra breve introducción al Q-Learning, conocemos una herramiena que si puede ayudarnos, las redes neuronales que sirven como funciones aproximadoras universales, entonces simplemente necesitamos entrenar una red que se asemeje a $Q^*$.

Para la regla de actualización de entrenamiento, usaremos el hecho de que cualquir función $Q$ para alguna táctica obedece a la ecuación de Bellman:

$$Q^{\pi}(s, a) = r + \gamma Q^{\pi}(s', \pi(s'))$$

La diferenia entre ambos lados de la ecuación es conocida como el error de diferencia temporal, $\delta$:

$$\delta = Q(s, a) - (r + \gamma\ max\ Q(s', a))$$

### Definición

Para la clase `Trainer` tenemos dos métodos:

- El método `generate_trajectories` que se encarga de generar trayectorías y calcular la recompensa en base a las acciones.
- El método es el `update_q_net` que se encarga actualizar la red por medio de aproximaciones de la función $q$ usando la ecuación de *Bellman*.

```
class Trainer:
    @staticmethod
    def generate_trajectories(
        env: BaseEnv, q_net: VisualQNetwork, buffer_size: int, epsilon: float
    ):
        """
        Dado un Unity Environment y una Q-Network, este método generará un búfer de
```

```
        Experiencias obtenidas al ejecutar el Environment con la Policy derivada de
        la Q-Network.
        :param BaseEnv: El UnityEnvironment usado.
        :param q_net: La Q-Network usada para recolectar la data.
        :param buffer_size: El tamaño mínimo del buffer que devolverá este método.
        :param epsilon: Agregará una variable normal aleatoria con desviación estándar.
        epsilon a los valores de la Q-Network para fomentar la exploración.
        :returns: una Tuple que contiene el búfer creado y el promedio acumulado de los
        agentes obtenidos.
        """
        # Crear un buffer vacio.
        buffer: Buffer = []

        # Reiniciar el environment.
        env.reset()
        # Leer y guardar el nombre del comportamiento del env.
        behavior_name = list(env.behavior_specs)[0]
        # Leer y guardar las especificaciones del comportamiento del env.
        spec = env.behavior_specs[behavior_name]

        # Mapping AgentID -> trajectories. Ayuda a crear trayectorias para cada agente.
        dict_trajectories_from_agent: Dict[int, Trajectory] = {}
        # Mapping AgentId -> "last observation".
        dict_last_obs_from_agent: Dict[int, np.ndarray] = {}
        # Mapping AgentId -> "last action".
        dict_last_action_from_agent: Dict[int, np.ndarray] = {}
        # Mapping AgentId -> cumulative reward (Solo para reporte).
        dict_cumulative_reward_from_agent: Dict[int, float] = {}
        # Lista que guarda las recompensas acumuladas hasta el momento.
        cumulative_rewards: List[float] = []

        # Mientras no exista suficiente data en el buffer.
        while len(buffer) < buffer_size:
            # Obtener los Decision Steps y Terminal Steps del agente.
            decision_steps, terminal_steps = env.get_steps(behavior_name)

            # Para todos los agentes con un Terminal Step:
            for agent_id_terminated in terminal_steps:
                # Se crea la última experiencia.
                last_experience = Experience(
                      obs=dict_last_obs_from_agent[agent_id_terminated].copy(),
                      reward=terminal_steps[agent_id_terminated].reward,
                      done=not terminal_steps[agent_id_terminated].interrupted,
                      action=dict_last_action_from_agent[agent_id_terminated].copy(),
                      next_obs=terminal_steps[agent_id_terminated].obs[0],
                )
                # Se limpia la última observación y la última acción. (La trayectoria
termino).
                dict_last_obs_from_agent.pop(agent_id_terminated)
                dict_last_action_from_agent.pop(agent_id_terminated)
                # Se reporta la recompensa acumulada.
```

```
            cumulative_reward = (
              dict_cumulative_reward_from_agent.pop(agent_id_terminated)
              + terminal_steps[agent_id_terminated].reward
            )
            cumulative_rewards.append(cumulative_reward)
            # Se añade la Trayectoria y la última experiencia al buffer.
            buffer.extend(dict_trajectories_from_agent.pop(agent_id_terminated))
            buffer.append(last_experience)

        # Para todos los agentes con Decision Step:
        for agent_id_decisions in decision_steps:
            # Si el agente no tiene una trayectoria, se crea una vacia.
            if agent_id_decisions not in dict_trajectories_from_agent:
                dict_trajectories_from_agent[agent_id_decisions] = []
                dict_cumulative_reward_from_agent[agent_id_decisions] = 0

            # Si el agente solicita una decisión con la última observación
            if agent_id_decisions in dict_last_obs_from_agent:
                # Se crea una experiencia de la última observación y el Decision Step.
                exp = Experience(
                    obs=dict_last_obs_from_agent[agent_id_decisions].copy(),
                    reward=decision_steps[agent_id_decisions].reward,
                    done=False,
                    action=dict_last_action_from_agent[agent_id_decisions].copy(),
                    next_obs=decision_steps[agent_id_decisions].obs[0],
                )
                # Se actualiza la trayectoria del agente y su recompensa acumulada.
                dict_trajectories_from_agent[agent_id_decisions].append(exp)
                dict_cumulative_reward_from_agent[agent_id_decisions] += (
                    decision_steps[agent_id_decisions].reward
                )
            # Guarda la observación como la nueva última observación.
            dict_last_obs_from_agent[agent_id_decisions] = (
              decision_steps[agent_id_decisions].obs[0]
            )

        # Se genera una acción para cada agente que solicita una decisión.
        # Se calculan los valores para cada acción dada la observación.
        actions_values = (
            q_net(torch.from_numpy(decision_steps.obs[0])).detach().numpy()
        )
        # Se añade algo de ruido epsilon a los valores.
        actions_values += epsilon * (
            np.random.randn(actions_values.shape[0], actions_values.shape[1])
        ).astype(np.float32)
        # Selecciona la mejor acción usando argmax.
        actions = np.argmax(actions_values, axis=1)
        actions.resize((len(decision_steps), 1))
        # Guarda la acción, se pondrá en una trayectoría después.
        for agent_index, agent_id in enumerate(decision_steps.agent_id):
            dict_last_action_from_agent[agent_id] = actions[agent_index]
```

```
            # Se establecen las acciones del environment.
            # Los Unity Environments esperan instancias del tipo ActionTuple.
            action_tuple = ActionTuple()
            action_tuple.add_discrete(actions)
            env.set_actions(behavior_name, action_tuple)
            # Se realiza un paso en la simulación.
            env.step()
        return buffer, np.mean(cumulative_rewards)

    @staticmethod
    def update_q_net(
        q_net: VisualQNetwork,
        optimizer: torch.optim,
        buffer: Buffer,
        action_size: int
    ):
        """
        Realiza una actualización de Q-Network utilizando el optimizador y el búfer
proporcionados
        """
        BATCH_SIZE = 1000
        NUM_EPOCH = 3
        GAMMA = 0.9
        batch_size = min(len(buffer), BATCH_SIZE)
        random.shuffle(buffer)
        # Se separa el buffer en batches.
        batches = [
          buffer[batch_size * start : batch_size * (start + 1)]
          for start in range(int(len(buffer) / batch_size))
        ]
        for _ in range(NUM_EPOCH):
            for batch in batches:
                # Create the Tensors that will be fed in the network
                obs = torch.from_numpy(np.stack([ex.obs for ex in batch]))
                reward = torch.from_numpy(
                  np.array([ex.reward for ex in batch], dtype=np.float32).reshape(-1, 1)
                )
                done = torch.from_numpy(
                  np.array([ex.done for ex in batch], dtype=np.float32).reshape(-1, 1)
                )
                action = torch.from_numpy(np.stack([ex.action for ex in batch]))
                next_obs = torch.from_numpy(np.stack([ex.next_obs for ex in batch]))

                # Use the Bellman equation to update the Q-Network
                target = (
                  reward
                  + (1.0 - done)
                  * GAMMA
                  * torch.max(q_net(next_obs).detach(), dim=1, keepdim=True).values
                )
```

```
                mask = torch.zeros((len(batch), action_size))
                mask.scatter_(1, action, 1)
                prediction = torch.sum(qnet(obs) * mask, dim=1, keepdim=True)
                criterion = torch.nn.MSELoss()
                loss = criterion(prediction, target)

                # Perform the backpropagation
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
```

---

## 6. Entrenamiento.

Finalmente hemos definido todo lo que nuestro problema necesita, conocemos las definiciones, la estructura del proyecto, las funciones para comunicar Python con Unity, tenemos los modelos para el entrenamiento con redes neuronales... entonces, ¿qué falta?. ¡El entrenamiento por su puesto!

Primero intentamos cerra el entorno por si hay alguno; después creamos un nnuevo entorno e imprimimos en pantalla un mensaje para saber que el entorno fue creado y se conecto con Unity satisfactoriamente.

Creamos una instancia de nuestra VisualQNetwork con los parámetros de nuestro problema, es decir:

- `qnet_input = (84,84,4)` : por las 4 imagenes de 84×84 pixeles.
- `qnet_output = 3` : por el número de acciones que puede tomar el agente.
- `qnet_encoding = 126` : el número de neuronas que tendran las capas densas.

Creamos un buffer de experiencias, definimos el optimizador (Adam), la lista de las recompensas acumuladas.

y otros datos generales como el número pasos de entrenamiento, el número de experiencias y el tamañoi del buffer.

Para cada paso:

1. Creamos una nueva experiencia usando el método `generate_trajectories` de la clase `Trainer` con un epsilon de `0.1` .
2. Ordenamos de forma aleatoria las experiencias.
3. Ajustamos la lista de experiencias y añadimos la nueva experiencia.
4. Actualizamos la `qnet` con el método `update_q_net` de la clase `Trainer` usando las experiencias y el optimizador Adam.
5. Obtenemos las recompensas usando el método `generate_trajectories` de la clase `Trainer` con un epsilon de `0` .
6. Añadimos las recompensas a la lista de recompensas acumuladas.
7. Imprimimos los resultados.

finalmente cerramos en entorno con el comando `close()` e imprimirmos los resultados del entrenamiento.

```
# Cierra un env si no ha sido cerrado antes.
try:
```

```
    env.close()
except:
    pass

env = UnityEnvironment(file_name = None, base_port=5004)
print("Environment created")

# Se crea una Q-Network.
qnet_input = (84,84,4)
qnet_output = 3
qnet_encoding_size = 126
qnet = VisualQNetwork(qnet_input, qnet_encoding_size, qnet_output)

# Se crea un bufer para las experiencias.
experiences: Buffer = []

# Se define el optimizador (Adam).
optim = torch.optim.Adam(qnet.parameters(), lr= 0.001)

# Contenedor para las recompensas acumuladas.
cumulative_rewards: List[float] = []

# Se definen el número de steps a realizar (70).
NUM_TRAINING_STEPS = 50

# Se define el número de experiencias a recolectar en cada step de entrenamiento.
NUM_NEW_EXP = 1000

# Se define el tamaño máximo del buffer.
BUFFER_SIZE = 10000

for n in range(NUM_TRAINING_STEPS):
    new_exp,_ = Trainer.generate_trajectories(env, qnet, NUM_NEW_EXP, epsilon = 0.1)
    random.shuffle(experiences)
    if len(experiences) > BUFFER_SIZE:
        experiences = experiences[:BUFFER_SIZE]
    experiences.extend(new_exp)
    Trainer.update_q_net(qnet, optim, experiences, 3)
    _, rewards = Trainer.generate_trajectories(env, qnet, 100, epsilon = 0)
    cumulative_rewards.append(rewards)
    print("Training step", n+1, "\treward", rewards)

env.close()
print("Closed environment.")

# Muestra el gráfico de entrenamiento.
plt.plot(range(NUM_TRAINING_STEPS), cumulative_rewards)
```

# Referencias.


1. Technologies, U. (2020, 5 junio). Unity - Manual: Unity User Manual (2019.3). Unity 2019 - Documentation. https://docs.unity3d.com/2019.3/Documentation/Manual/index.html
2. U. (2020). Unity-Technologies/ml-agents. GitHub ML-Agents Toolkit. https://github.com/Unity-Technologies/ml-agents
3. U. (2020). Unity-Technologies/ml-agents. GitHub ML-Agents Python API. https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Python-API-Documentation.md
4. Juliani, A., Berges, V., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., Lange, D. (2020). Unity: A General Platform for Intelligent Agents. arXiv preprint arXiv:1809.02627. https://github.com/Unity-Technologies/ml-agents.
5. Kenney • Racing Kit. (2010). Kenney Racing Kit. https://www.kenney.nl/assets/racing-kit
6. Reinforcement Learning Series Intro - Syllabus Overview. (2018). Deeplizard. https://deeplizard.com/learn/video/nyjbcRQ-uQ8
7. Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning, Second Edition: An Introduction (2nd ed.) [Libro electrónico]. Bradford Book. http://incompleteideas.net/book/RLbook2020.pdf