

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL



VNIVERSITAT
ID VALÈNCIA

TRABAJO DE FIN DE GRADO

**DESARROLLO DE UN SIMULADOR DE CONDUCCIÓN
AUTÓNOMA CON UNITY 3D Y ROS PARA USO DE
RED NEURONAL ARTIFICIAL**

AUTOR/A:

SERGIO JOSÉ PÉREZ RIOS

TUTORÍA:

VALERO LAPARRA PÉREZ-MUELAS

VICENT GIRBÉS JUAN

SEPTIEMBRE, 2020

Agradecimientos

Este ha sido un camino difícil y no solo es mérito mío, en el trayecto muchas personas me han ayudado a conseguirlo y por tanto quiero dedicar este espacio para agradecerles por todo, primero que nada, mis padres quienes han sido el pilar para mantenerme con fuerza y ganas a pesar de todo lo que he tenido que vivir para llegar hasta aquí, a mis hermanas y familiares que me han dado su apoyo para superar dificultades.

A compañeros de trabajo y amigos que han apoyado y ayudado en sus posibilidades y siempre han estado allí Paula Vasileva, Esther Debón , Jose Luis Irizarri y Yennireth Palma.

Mis tutores que sin ellos no fuese logrado esto, ni descubierto mi pasión por esta rama de la ingeniería que me motiva a seguir formándome como profesional.

A esas personas que encontré en el camino, a mis amigos Jorge Martínez, Sarah Camantigue, David Tomas y Melyssa Navarro. Y por supuesto a mi amiga y compañera tutorías Eveline Mihut que con sus conocimientos, paciencia y apoyo ha sido de mucha ayuda en este último tramo del proyecto. Tanto ella como los cuatro mencionados antes han hecho estos 4 años una experiencia inolvidable.

A todos Muchas Gracias

Resumen

El objetivo de este proyecto es realizar un simulador en la plataforma Unity 3D que pueda ser usado por una red neuronal que implemente conducción autónoma. Como base se utiliza un código desarrollado en Python para el control, el entorno gráfico en Unity 3D, ROS (Robot Operating System) para la comunicación entre Unity y la red neuronal, y RosSharp para la comunicación entre Unity y ROS. Por último, se utilizará openCV para la realización de la red neuronal y el tratamiento de las imágenes. Para ello se realizará el modelado de un entorno virtual 3D similar al que ha sido utilizado para entrenar la red neuronal.

El modelo del robot será capaz de captar su entorno con una cámara y enviárselo mediante ROS a la red neuronal artificial. Esto servirá para poder utilizar la información tanto en fase de entrenamiento como en fase de test. Lo cual permitiría entrenar una red neuronal artificial que fuera capaz de tomar decisiones dependiendo de si hay una curva, intersección para girar o un obstáculo para detenerse, realizando así una conducción autónoma.

Resum

L'objectiu d'aquest projecte és la realització d'un simulador en la plataforma Unity 3D per poder ser utilitzat per una xarxa neuronal que implementa conducció autònoma. Con a base s'utilitza un codi desenvolupat en Python per al control, l'entorn gràfic en Unity 3D, ROS (Robot Operating System) per a la comunicació entre Unity i la xarxa neuronal, i RosSharp per a la comunicació entre Unity i ROS. Per últim, s'utilitzarà openCV per a la realització de la xarxa neuronal i el tractament d'imatges. Per a aconseguir-ho, es realitzarà el modelat d'un entorn virtual 3D similar a l'utilitzat per entrenar la red neuronal.

El model del robot serà capaç de captar el seu entorn mitjançant una càmera i enviar-ho a la red neuronal a través de ROS. Açò servirà per poder utilitzar la informació tant en la fase d'entrenament com en la fase de test, amb la finalitat d'entrenar una xarxa neuronal artificial que fóra capaç de prendre decisions depenent de si hi ha una corva, intersecció per a girar o un obstacle per detindre's, realitzant així una conducció autònoma.

Contenido

1	Introducción.....	7
2	Conceptos y lenguajes de programación utilizados.....	8
3	Esquema	10
3.1	El entorno:.....	11
3.2	El canal:.....	11
3.3	El cerebro:	11
4	Bloque 1 – Entorno Unity 3D.....	12
4.1	Instalación y configuración básica	12
4.2	“Assets”.....	13
4.2.1	“Low Poly Destructible 2Cars no.8”	14
4.2.2	“Quantum Theory”	14
4.2.3	“Standard Assets”	14
4.2.4	Tfg_Scripts.....	14
4.2.5	ROS#.....	14
4.3	Desarrollo del entorno.....	15
4.3.1	Construcción de la ciudad.....	15
4.3.2	Creación del Terreno	16
4.3.3	Ensamblado del modelo del Coche.....	16
4.3.4	ROS conector	23
5	Bloque 2 – ROS / ROS# y RosBridge	25
5.1	Máquina virtual (VM), con sistema operativo Ubuntu 18.04	25
5.2	Instalación de ROS Melodic	26
5.3	Interfaz Gráfico	29
5.4	Configuración de Paquetes.....	30
5.4.1	Configuración del Paquete TFG-Scripts.....	30
5.4.2	Configuración de los Paquetes de Ros#.....	30
5.5	Comunicación entre entorno Unity y control en Python.....	31
5.5.1	Topics Utilizados	31
5.5.2	Mensajes Utilizados	31
5.5.3	Scripts de Comunicación por ROS# (Nodos).....	32
5.6	Red de comunicación	37
6	Bloque 3 – Script de control en Python y Red neuronal	39
6.1	Segmento 1: RCDriving	39

6.2	Segmento 2: <i>Collect data training</i>	42
6.3	Segmento 3: función <i>main</i>	46
6.4	Segmento 4: encabezado	46
7	Orden de ejecución.	47
8	Resultados.....	49
8.1	Modo entrenamiento.	50
8.2	Modo Simulación	53
9	Conclusiones.....	56
10	Referencias	58
11	ANEXOS.....	59
11.1	Enlaces GitHub	60
11.2	Código entero Python.....	60

Lista de Figuras

Figura 1, Perceptrón Multicapa.	9
Figura 2, Esquema con la distribución del simulador.....	11
Figura 3, Configuración del proyecto en Unity.	13
Figura 4, “Assets” implementados.....	13
Figura 5, Construcción de la ciudad en Unity	15
Figura 6, Creación del Terreno Unity.....	16
Figura 7, Componente Wheel Collider.....	17
Figura 8, Objeto Cámara.....	18
Figura 9, Script control cámara.....	19
Figura 10, Script Control Coche Simulación.....	20
Figura 11, Script Control Coche Funciones.....	21
Figura 12, Script Control Coche Funciones y Declaración de Variables.	22
Figura 13, Configuración Componentes Coche.....	23
Figura 14, Objeto ROS Conector.....	24
Figura 15, Características de Windows, subsistema para Linux.	25
Figura 16, Máquina virtual Ubuntu 18.04 LTS Microsoft Store.....	26
Figura 17, Terminal Ubuntu	26
Figura 18, Comandos para instalación ROS Melodic.....	27
Figura 19, Configuración Paquetes ROS Melodic.....	28
Figura 20, Comandos para creación catkin workspace.....	28
Figura 21, Servidor Xming	29
Figura 22, Vista del script dentro del paquete creado.....	30
Figura 23, Script Twist Subscriber.	33
Figura 24, Script Image Publisher.	34
Figura 25, Script Joy Publisher Original.	35
Figura 26, Script Joy Publisher Mine parte I.	36
Figura 27, Script Joy Publisher Mine parte II.....	37
Figura 28, Red de flujo de información entre nodos y Topics (modo entrenamiento).....	38
Figura 29, Red de flujo de información entre nodos y Topics (modo simulación).	38
Figura 30, Clase RCControl.....	40
Figura 31, Clase RCDriveNNOnly.....	41
Figura 32, Clase Red Neuronal.....	42
Figura 33, Clase CollectDataTraining parte I.....	43
Figura 34, Clase CollectDataTraining parte II.....	44
Figura 35, Función load_data.	45
Figura 36, Función entrenar.....	45
Figura 37, Programa principal.	46
Figura 38, Encabezado inclusión de librerías.	46
Figura 39, lanzamiento de rosbriidge.....	47
Figura 40, Simulador en marcha.....	48
Figura 41, Conexión a rosbriidge.....	48
Figura 42, Diagrama de flujo funcionamiento script.....	50
Figura 43, Archivos .npz guardados.	51
Figura 44, Modo entrenamiento en ejecución.	52
Figura 45, Modelo original de zhaoying9105.....	53

Figura 46, Simulación Red no entrenada.....	54
Figura 47, Simulador en modo simulación con red entrenada.	55

1 Introducción

El avance de las tecnologías en los últimos años ha hecho que hablar de conducción autónoma no parezca tan futurista, aunque requiere de muchos factores para lograrlo. Influyen sensores, como cámaras y localización GPS, inteligencia artificial y muchas otras herramientas que ya han sido incorporadas en los vehículos. Claro está que para ello aún queda un tramo por descubrir, tanto en el ámbito tecnológico como en temas de seguridad vial al implementar vehículos sin conductor. Sin embargo, ya no es algo que esté a un futuro muy lejano.

Si bien para el funcionamiento de la conducción autónoma el conjunto de tecnologías mencionadas anteriormente es importante como partes de un todo, la que se consideraría de mayor importancia es la red neuronal utilizada. Cuanto más compleja y mejor entrenada más acertadas y precisas serán las decisiones que tome y, por ende, la conducción autónoma.

En este trabajo se va a desarrollar un simulador en el que se puedan entrenar y validar los algoritmos de inteligencia artificial, el cual también será capaz de controlar un vehículo.

2 Conceptos y lenguajes de programación utilizados

Antes de comenzar es importante tener claros los conceptos a tratar y dar una breve explicación de cada uno de ellos. Así como mencionar los lenguajes de programación implementados.

Red Neuronal Artificial (RNA): Son redes conformadas por neuronas artificiales interconectadas entre sí y organizadas de manera jerárquica. Poseen un valor numérico modificable llamado peso, que determinan la intensidad de la señal de entrada registrada por una neurona artificial. Las RNA intentan interactuar con objetos externos tomando datos de entrada pasándolos por sus capas internas y luego dando un resultado al exterior según la información que ellas tengan aprendidas realizando una predicción para dicho valor de entrada.

La actividad que una unidad de procesamiento o neurona artificial realiza en un sistema de este tipo es simple. Normalmente, consiste en sumar los valores de las entradas (inputs) que recibe de otras unidades conectadas a ella, comparar esta cantidad con el valor umbral y, si lo iguala o supera, enviar activación o salida (output) a las unidades a las que esté conectada. Tanto las entradas que la unidad recibe como las salidas que envía dependen a su vez del peso o fuerza de las conexiones por las cuales se realizan dichas operaciones.

El aprendizaje en una RNA es un proceso de ajuste o modificación de los valores o pesos de las conexiones, en el que se logra que las salidas del sistema sean lo más parecidas a las salidas deseadas proporcionadas por el usuario. Las redes neuronales artificiales pueden clasificarse de acuerdo con el tipo de aprendizaje que utilizan.

El modelo de RNA que se va a implementar en este proyecto es del tipo Perceptrón Multicapa MLP. El cual está formado por una capa de entrada una de salida y al menos una capa oculta como vemos en la Figura 1.

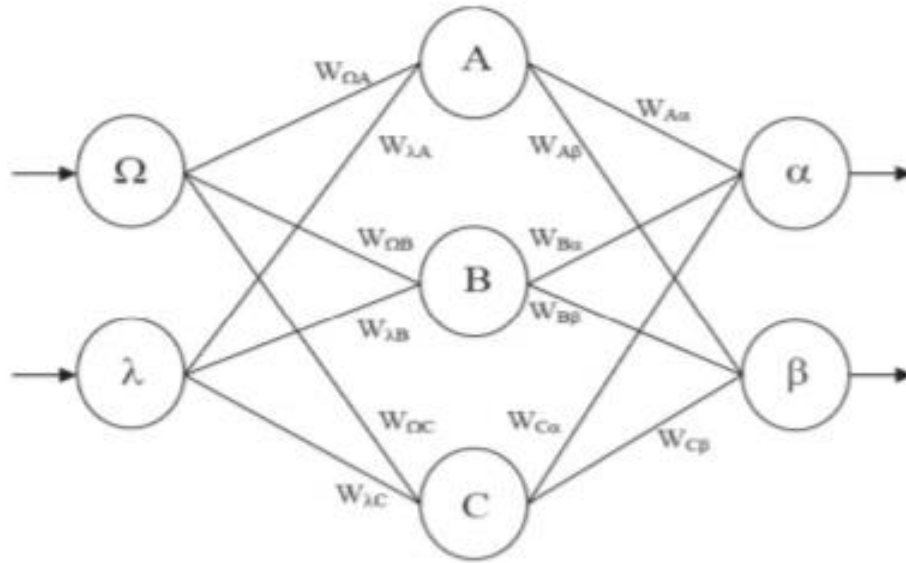


Figura 1, Perceptrón Multicapa.

Dentro de las características más importantes del perceptrón multicapa se encuentran las siguientes:

- Se trata de una estructura altamente no lineal.
- El sistema es capaz de establecer una relación entre dos conjuntos de datos.
- Existe la posibilidad de realizar una implementación hardware.

El algoritmo de entrenamiento consiste en utilizar el error generado por la red y propagarlo hacia atrás, es decir, reproducirlo hacia las neuronas de las capas anteriores. Para el entrenamiento de una red hemos de tener en cuenta que la salida de cada neurona no va a depender únicamente de las entradas del problema, sino que también depende de las salidas que ofrezcan el resto de las neuronas. Por este mismo motivo también podemos afirmar que el error cometido por una neurona no solo va a depender de que sus pesos sean los correctos o no, sino que dependerá del error que traiga acumulado del resto de neuronas que le precedan en la red.

Para controlar el error cometido se ha de redefinir la función de error, de tal forma que la función de error es la siguiente:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{salidas}} (t_{kd} - o_{kd})^2$$

Ecuación 1, Ecuación de función de error

Donde cada parámetro significa lo siguiente

- \vec{w} es el vector de pesos.
- D es el conjunto de ejemplos de entrenamiento.
- d es un ejemplo de entrenamiento concreto.

- Salidas es el conjunto de neuronas de salida.
- k es una neurona de salida.
- t_{kd} es la salida correcta que debería dar la neurona de salida k al aplicarle a la red el ejemplo de entrenamiento d .
- o_{kd} es la salida que calcula la neurona de salida k al aplicarle a la red el ejemplo de entrenamiento d .

ROS (Robot Operating System): es un firmware flexible que permite el desarrollo software de los robots, dotando de múltiples herramientas, librerías y convenios que tienen como finalidad simplificar las tareas a la hora de crear un comportamiento robótico para distintas plataformas.

Cabe destacar las principales características que posee y explicarlas brevemente.

- **Paquetes:** El software en ROS está organizado en paquetes. Un paquete puede contener un nodo, una librería, conjunto de datos, o cualquier cosa que pueda constituir un módulo.
- **Nodos:** es un proceso que realiza algún tipo de computación en el sistema. Los nodos se combinan dentro de un grafo, compartiendo información entre ellos, para crear ejecuciones complejas.
- **Topic:** Son canales de información entre los nodos. Un nodo puede emitir o suscribirse a un tópico.
- **Mensajes:** es una estructura de datos que están comprimidos en una serie de campos, estos pueden ser desde simples tipos de datos (int, float, char), hasta los más complejos como arrays, vectores o matrices. Así como también pueden ser un conjunto de estructuras. Por convención los mensajes tienen por nombre “nombre del paquete + nombre del mensaje + .msg”.

Conducción autónoma:

Es una modalidad de conducción que consiste en el manejo de un vehículo sin el control activo de un conductor. Un vehículo autónomo está considerado como tal cuando está equipado con la suficiente tecnología que permita su manejo sin precisar de forma activa el control o supervisión de un conductor, tanto si dicha tecnología autónoma estuviera activa o desactivada, de forma permanente o temporal.

Lenguajes de Programación:

- C# para scripts en Unity y ROS#.
- Python para la RNA y el script de control en ROS.
- OpenCV para el procesamiento de las imágenes.

3 Esquema

El trabajo fue realizado mediante diversos elementos de programación que podemos dividir en 3 partes igual de importantes y donde cada una de ellas lleva su trabajo de configuración y de desarrollo. Para una comprensión rápida se explicará en un esquema lo más relevante de cada parte, y más adelante se detallará en profundidad cada una de ellas.

Antes que nada, hay que aclarar que se utiliza como sistema operativo base Windows 10 para el entorno en Unity y una máquina virtual con Ubuntu 18.04 donde se encuentra el script de Python con la RNA, además de la versión ROS Melodic para la comunicación. Con la peculiaridad que la máquina virtual es una extensión de Windows que conserva las funcionalidades de la terminal, pero no dispone de interfaz gráfico por ende se implementó un servidor “Xming”, que el cual permite la comunicación entre el subsistema operativo Linux con un display online para poder utilizarlo como pantalla de la máquina virtual.

En la Figura 2 se observa el funcionamiento básico del simulador. Se divide en El entorno conformado desarrollado en Unity 3D, el canal de comunicación ROS/ROS# y el cerebro que es la red neuronal implementada en un script en Python.

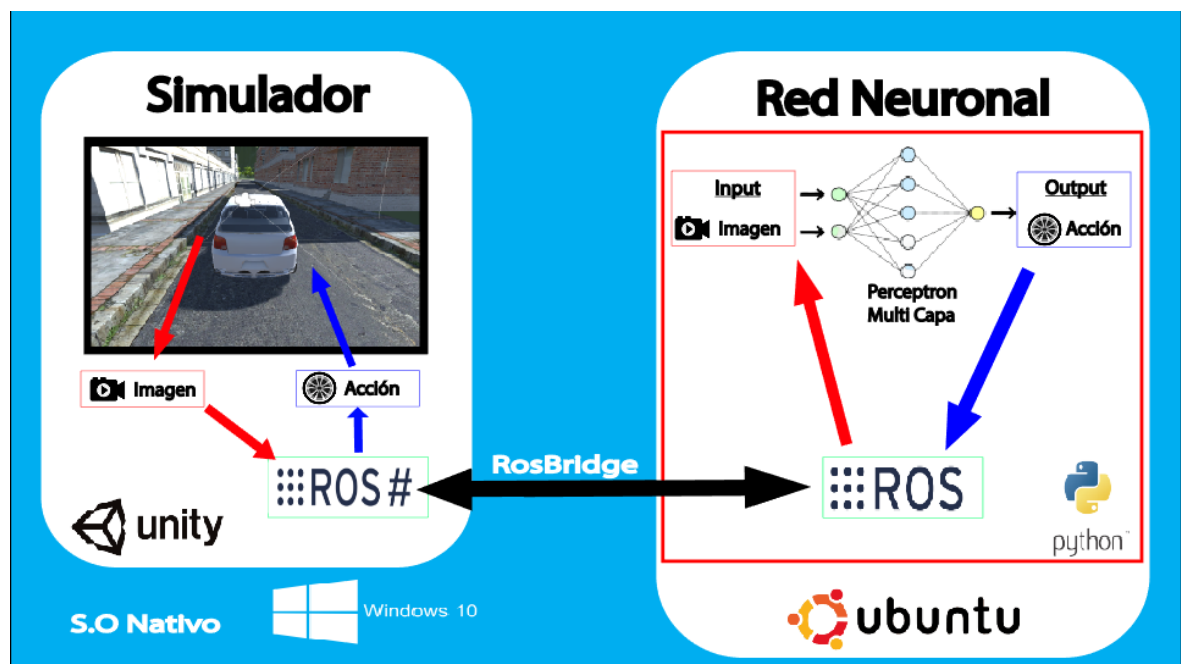


Figura 2, Esquema con la distribución del simulador.

3.1 El entorno:

Este es el ambiente, la parte física del simulador donde el usuario interactúa con él. Para su desarrollo se utilizó el programa Unity 3D, en el cual se ha modelado el entorno (carretera, vehículo, señal de STOP, ambiente, cámara), además se han realizado scripts dentro de Unity para poder mover el vehículo y la cámara.

3.2 El canal:

Es el encargado de obtener los datos del entorno (imagen, selector entrenar/simular) mediante ROS para enviárselos a la RNA que está en un script en Python, para que evalúe y haga la predicción o para entrenar a la misma. Y posteriormente ésta devuelva mediante ROS la dirección a la cual se debe mover o si debe mantener la velocidad o detenerse.

3.3 El cerebro:

Es donde se procesa toda la información recibida y se toman decisiones a partir de ella. Está conformado por un script de Python que a su vez está dividido en 2 partes entrenar/ simular,

dependiendo que señal esté activa se entrenará a la red para nuevos entornos o simulará el modelo que ya ha aprendido.

4 Bloque 1 – Entorno Unity 3D

Como se ha mencionado anteriormente es el interfaz gráfico con el que el usuario interactúa, por tanto, es el que describiremos en profundidad primero. Explicando desde la instalación, configuración de assets, desarrollo del entorno (modelado y scripting), funcionamiento y, por último, cómo se enviará/recibirá la información hacia/desde ROS.

4.1 Instalación y configuración básica

Descargar desde su sitio web oficial la versión de Unity 19.03f que es la compatible con ROS#. Se instalará Unity Hub para organizar los proyectos de mejor manera. Luego, al iniciar un nuevo proyecto configuramos las entradas como se observa en la Figura 3, que son las que implementamos en este proyecto. Utilizaremos los ejes Horizontal, Vertical, Train (que fue modificado se llamaba Fire 1) y, por último, Jump que es el que utilizaremos para frenar.

W → Adelante

A → Izquierda

T → Entrenar/Simular

S → Atras

D → Derecha

Jump/space → Frenar

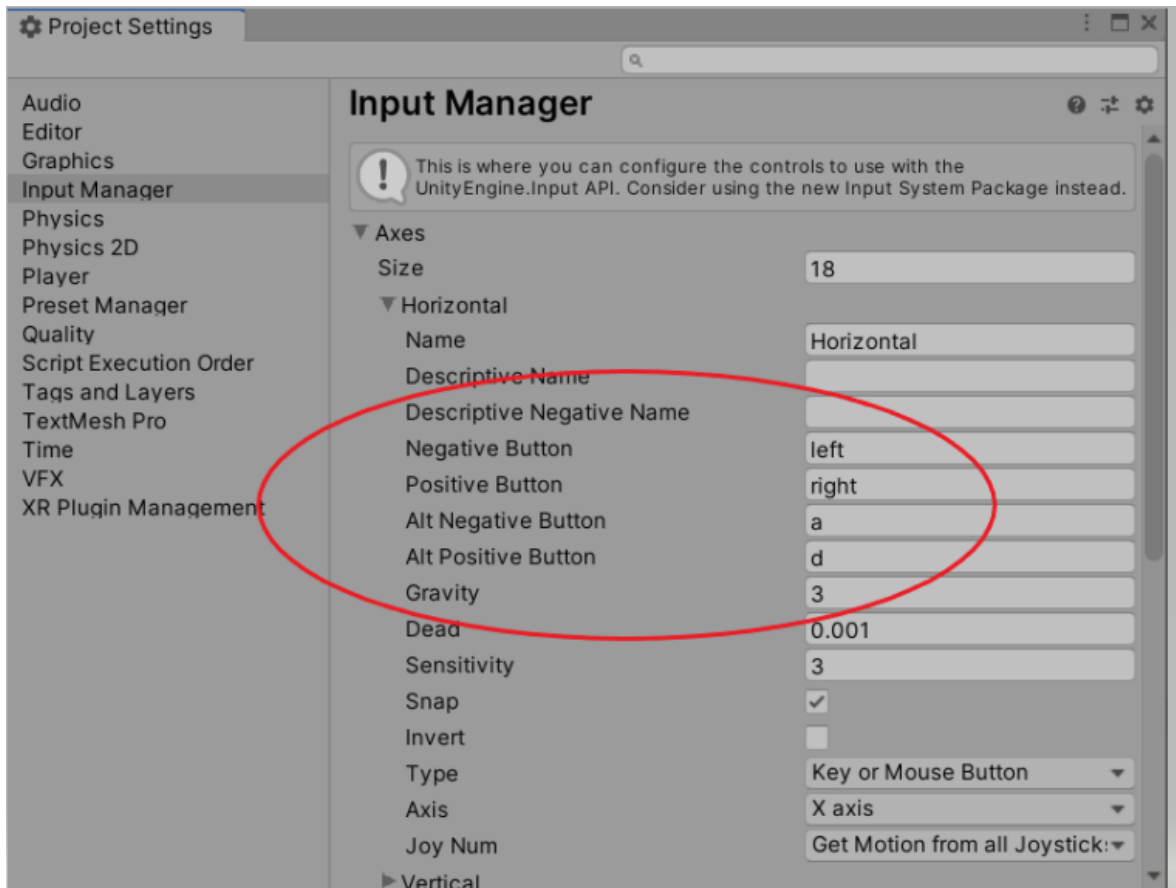


Figura 3, Configuración del proyecto en Unity.

4.2 “Assets”.

Un “Asset” es una representación de cualquier ítem que puede ser utilizado en el proyecto, puede ser desde modelos hasta comunicaciones. Éstos pueden ser descargados directamente desde la “AssetStore” de Unity. En la Figura 4 observamos los “assets” utilizados.

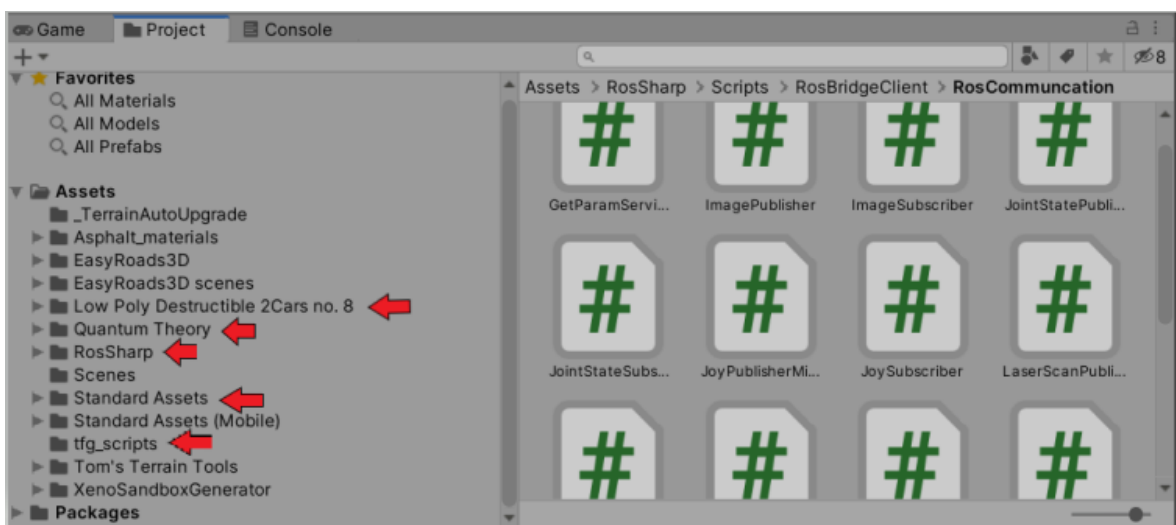


Figura 4, “Assets” implementados.

4.2.1 “Low Poly Destructible 2Cars no.8”

Éste dispone de "prefabs" (modelos prefabricados) y texturas del modelo del coche.

4.2.2 “Quantum Theory”

Dotado de "prefabs", texturas y scripts para la construcción de ciudades (edificios, pavimentos, aceras, puentes, etc.).

4.2.3 “Standard Assets”

Éste es básico del estándar, el cual permite modificar Terreno, HUD/UI mediante Canvas.

4.2.4 Tfg_Scripts

Es un “Asset” Personalizado en el cual almacenamos los scripts de control del coche y del control de la cámara principal.

4.2.5 ROS#

Es un "Asset" desarrollado por Siemens que permite la comunicación de Unity 3D con ROS, mediante scripts ya programados, que a su vez son nodos de ROS que pueden ser modificados para enviar cualquier tipo de mensaje soportado por ROS y mediante cualquier "Topic". También dispone de scripts que permiten modificar las características físicas en Unity, por ejemplo, la posición de una articulación ("hinge joints").

4.3 Desarrollo del entorno

4.3.1 Construcción de la ciudad.

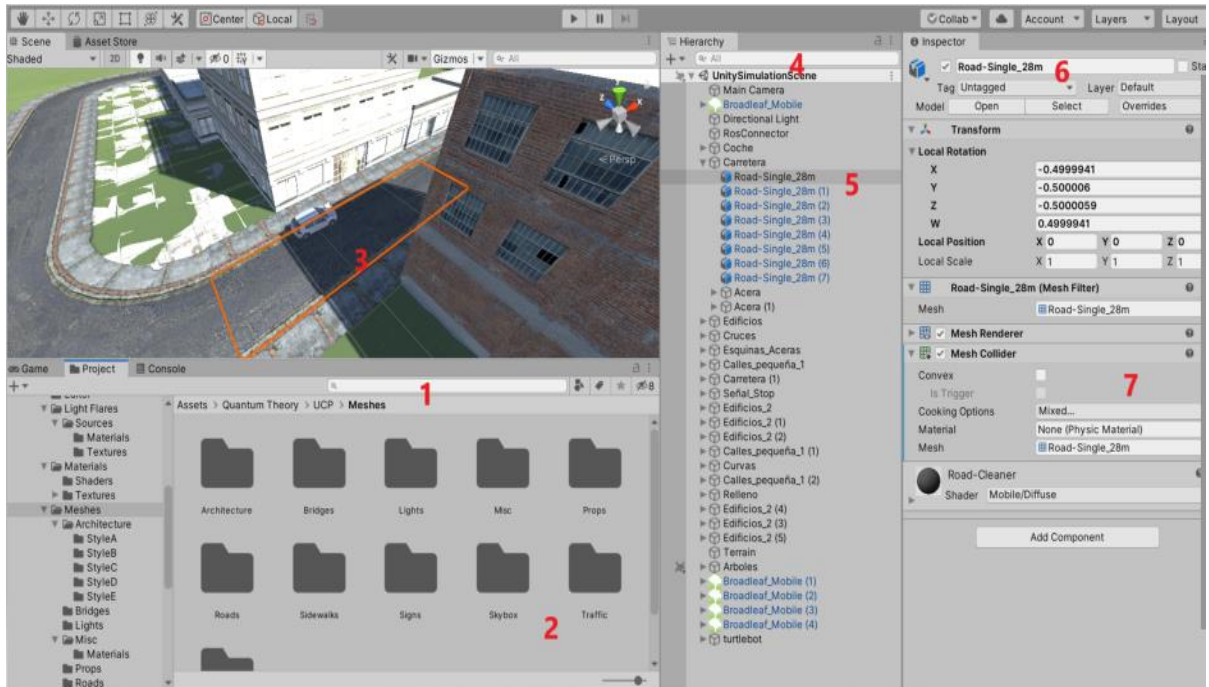


Figura 5, Construcción de la ciudad en Unity

En la Figura 5, se observan diferentes zonas de importancia a la hora de realizar la construcción de la ciudad.

- 1- Ruta donde se encuentran los "prefabs" en carpetas divididos por tipos de objeto.
- 2- Las diferentes carpetas que contienen los "prefabs" que utilizamos: para las calles los que se encuentran en "Roads", para las aceras en "sidewalks" y para los edificios los que se encuentran en "arquitectura".
- 3- Aquí se encuentra la escena "scene" donde desde las carpetas del proyecto podemos arrastrar o colocar directamente el objeto en el mundo, donde inicialmente tenemos como origen de coordenadas el (0,0,0).
- 4- En la ventana de Jerarquía tenemos todos los objetos que se van colocando en la escena.
- 5- Para tener mejor organización se pueden crear grupos. Como se observa, "carretera" engloba muchos trozos de caminos o "edificios" contienen múltiples edificios.
- 6- Ventana Inspector, aquí pueden visualizar todas las características de un objeto, así como añadir nuevas o agregar scripts para modificar dicho objeto o tener una interacción con otro.
- 7- Es muy importante que todos los modelos colocados en el mapa dispongan, como se ve en la Figura 5, de un Mesh collider porque así estos disponen de un detector de colisiones y ningún objeto con detector de colisiones pueda atravesarlo.

Siguiendo con los pasos anteriores, se construye la ciudad duplicando los objetos y concatenando los objetos, asegurando que no existan espacios vacíos entre ellos.

4.3.2 Creación del Terreno

Para modelar el terreno y las áreas verdes de la ciudad, se siguen los pasos que vemos en la Figura 6.

- 1- Se crea un Objeto del tipo “Terrain” en la jerarquía.
- 2- En las características se selecciona la textura y con el pincel para crear las montañas.
- 3- Se crean las montañas tirando del terreno creado en la ventana “scene”.
- 4- En la ruta del proyecto, se puede ver que disponemos de las texturas y de “prefabs” de árboles, arbustos y otros elementos de decoración.
- 5- Los “prefabs” colocados en la jerarquía con sus respectivos “mesh collider”.

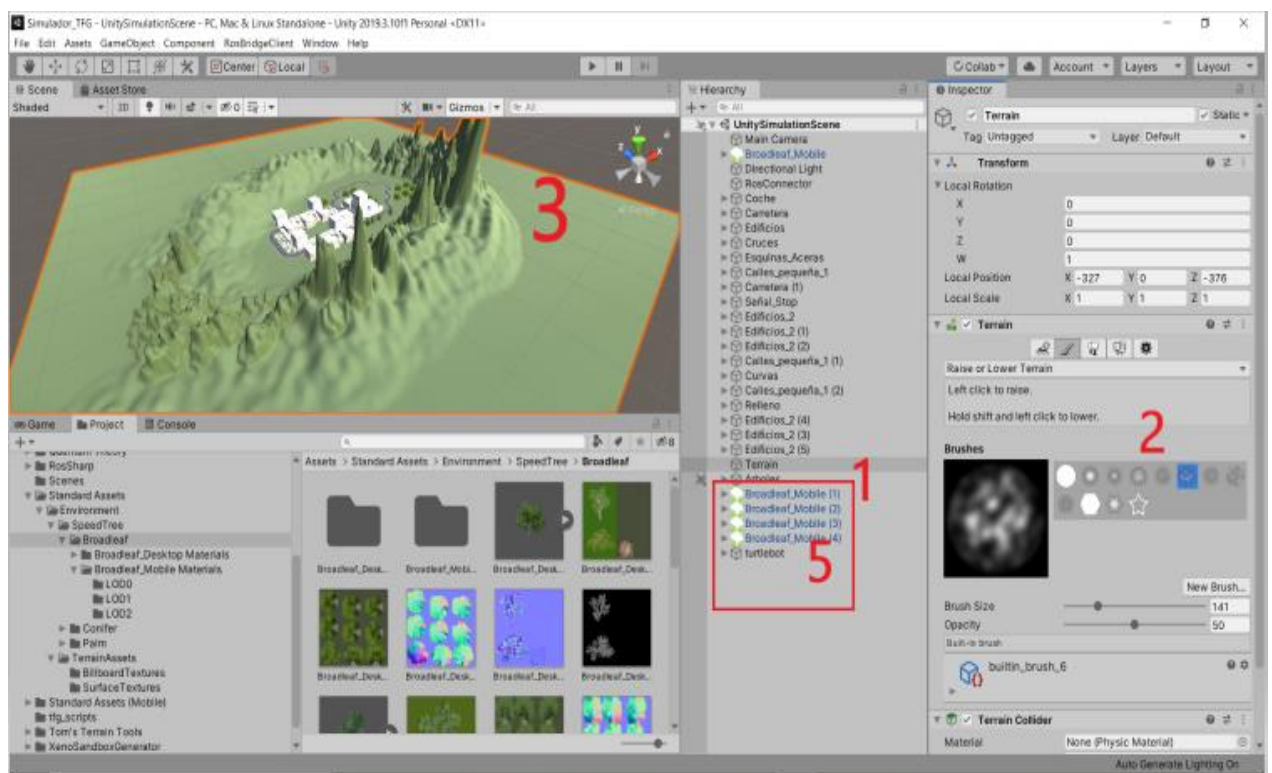


Figura 6, Creación del Terreno Unity.

4.3.3 Ensamblado del modelo del Coche

Para la construcción del modelo del coche se ensambla de igual forma que los edificios, seleccionando los “prefabs” desde el “asset” (“Low Poly Destructible 2Cars no.8”), con la peculiaridad de que a las ruedas hay que agregarle el componente “Wheel Collider” para que pueda interactuar como colisiones con el terreno. Se deben asignar los valores como observamos en la Figura 7, para que el movimiento sea con suspensión y adquiera velocidad

e inercia. Es importante que dentro de la jerarquía (“hierarchy”) las “WheelsCollider” estén por debajo del modelo del coche (“Classic_16”) porque la velocidad que adquieran los “colliders” de las ruedas se transmitirán al objeto padre.

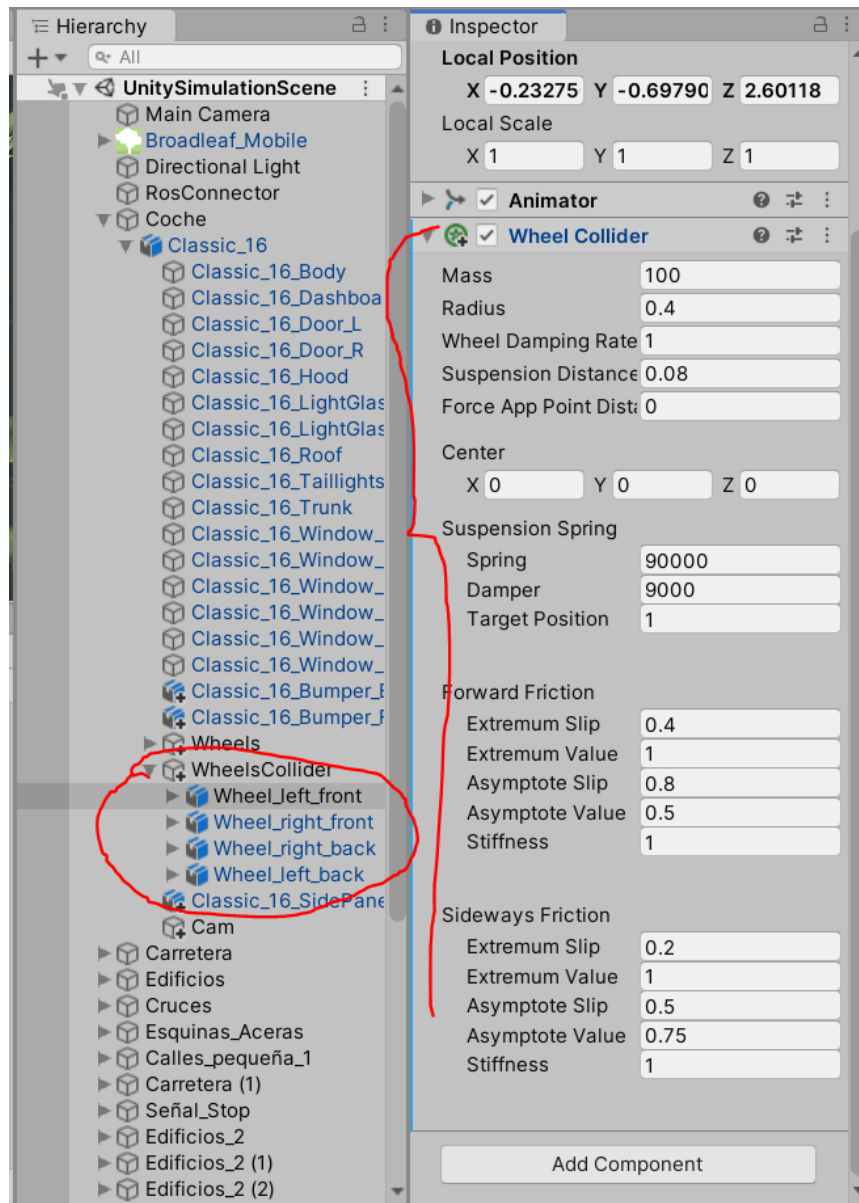


Figura 7, Componente Wheel Collider.

Por otro lado, al coche hay que asignarle un objeto tipo cámara, mediante el cual se adquiere la información que se pasa al ROS. En la Figura 8 se observa la configuración adaptada para dicho objeto.

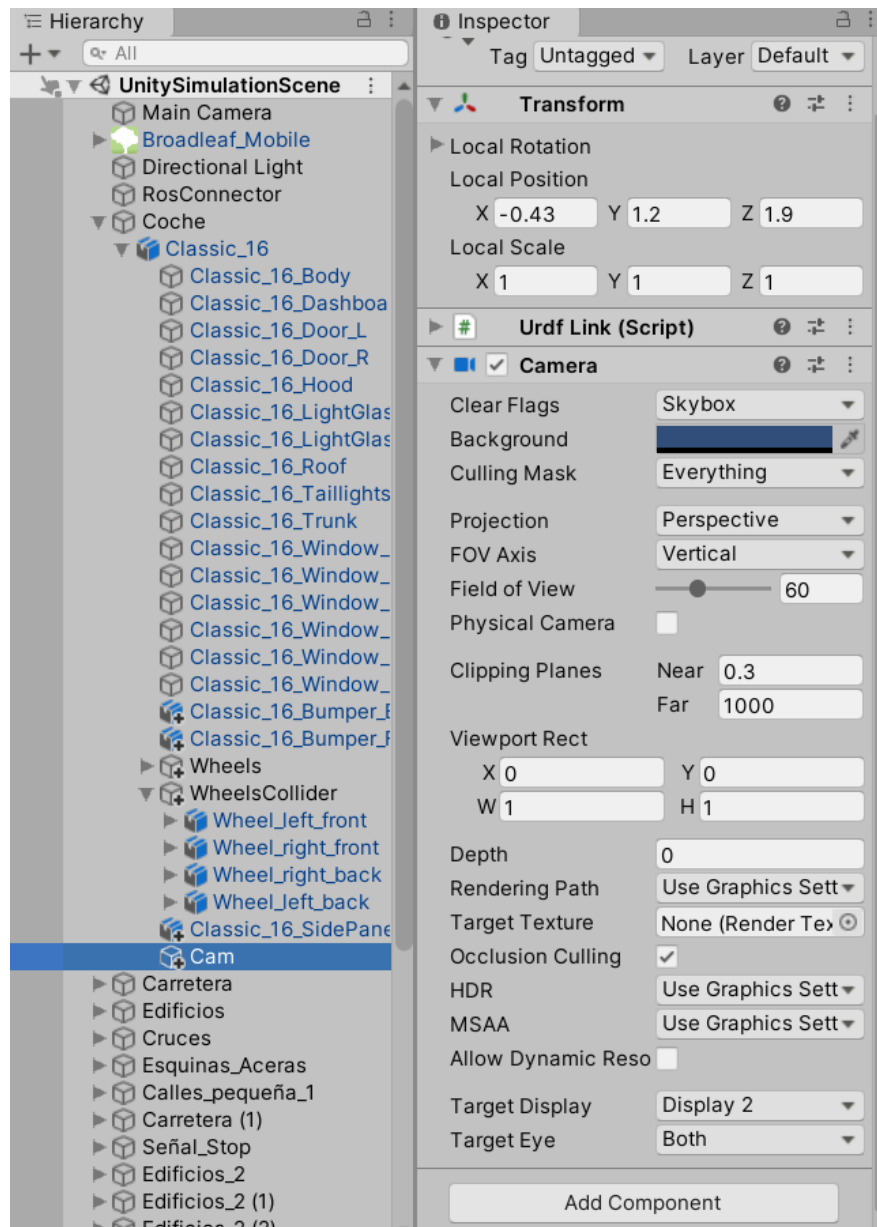


Figura 8, Objeto Cámara

Por último, hay que asignar al conjunto entero los scripts que se encargan de controlar el movimiento del coche en los dos modos (entrenamiento y simulación), además del script para controlar el movimiento de la cámara.

- Control Cámara:
 1. Declaración de variables.
 2. Es el Update() del loop que se ejecuta continuamente hasta que haya una interrupción. Aquí están las funciones lookAtTarget y MovetoTarget.
 3. Las dos funciones:
 - a. LookAtTarget: posiciona la cámara según donde esté mirando el objeto padre.
 - b. MoveToTarget: Sigue al objeto padre cuando se mueva en la dirección que está mirando.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Control_Cam_Principal : MonoBehaviour
6  {
7
8      public void LookAtTarget ()
9      {
10         Vector3 _lookDirection = objectToFollow.position - transform.position;
11         Quaternion _rot = Quaternion.LookRotation(_lookDirection, Vector3.up);
12         transform.rotation = Quaternion.Lerp(transform.rotation, _rot, lookSpeed * Time.deltaTime);
13     }
14
15     public void MoveToTarget()
16     {
17         Vector3 _targetPos = objectToFollow.position +
18                             objectToFollow.forward * offset.z +
19                             objectToFollow.right * offset.x +
20                             objectToFollow.up * offset.y;
21         transform.position = Vector3.Lerp(transform.position, _targetPos, followSpeed * Time.deltaTime);
22     }
23
24     private void FixedUpdate()
25     {
26         LookAtTarget();
27         MoveToTarget();
28     }
29
30
31
32     public Transform objectToFollow;
33     public Vector3 offset;
34     public float followSpeed = 10;
35     public float lookSpeed = 10;
36
37
38 }

```

Handwritten annotations in red: A bracket on the right side groups the `LookAtTarget` and `MoveToTarget` methods, with a circled '3' next to it. A circled '2' is placed next to the `FixedUpdate` method. A circled '1' is placed next to the public fields at the bottom.

Figura 9, Script control cámara.

- Control Coche Simulación:

En el control en modo simulación, las decisiones las toma el script de Python que implementa la red neuronal que se explicará más adelante. Este script en C# es el encargado de ejecutar las órdenes y por medio de ROS# recibe un mensaje de tipo Twist (comando de velocidad lineal y angular) por el topic /cmd_vel.

```

namespace RosSharp.RosBridgeClient
{
    public class TwistSubscriber : UnitySubscriber<MessageTypes.Geometry.Twist>
    {
        public Transform SubscribedTransform;

        private float previousRealTime;
        private Vector3 linearVelocity;
        private Vector3 angularVelocity;
        private bool isMessageReceived;

        protected override void Start()
        {
            base.Start();
        }

        protected override void ReceiveMessage(MessageTypes.Geometry.Twist message)
        {
            linearVelocity = ToVector3(message.linear).Ros2Unity();
            angularVelocity = -ToVector3(message.angular).Ros2Unity();
            isMessageReceived = true;
        }

        private static Vector3 ToVector3(MessageTypes.Geometry.Vector3 geometryVector3)
        {
            return new Vector3((float)geometryVector3.x, (float)geometryVector3.y, (float)geometryVector3.z);
        }

        private void Update()
        {
            if (isMessageReceived)
            {
                ProcessMessage();
            }
        }

        private void ProcessMessage()
        {
            float deltaTime = Time.realtimeSinceStartup - previousRealTime;

            SubscribedTransform.Translate(linearVelocity * deltaTime);
            SubscribedTransform.Rotate(Vector3.forward, angularVelocity.x * deltaTime);
            SubscribedTransform.Rotate(Vector3.up, angularVelocity.y * deltaTime);
            SubscribedTransform.Rotate(Vector3.left, angularVelocity.z * deltaTime);

            previousRealTime = Time.realtimeSinceStartup;

            isMessageReceived = false;
        }
    }
}

```

Figura 10, Script Control Coche Simulación.

En la Figura 10, encontramos el script que ya viene predefinido en el paquete de Asset de Rossharp, como “TwistSubscriber.cs”. Lo implementaremos tal cual para modificar el *Transform* de las ruedas delanteras, ya que con el mensaje tipo Twist que nos enviará el script principal de control es suficiente para trabajar en el movimiento lineal y rotacional de las ruedas delanteras.

También se observa cómo está estructurado el script, utilizando un espacio de Rossharp:

1. Un encabezado donde están las declaraciones de variables y la función de inicio Start.
2. Función de mensaje recibido de tipo Twist, donde se asigna el valor a las variables.
3. Función que genera un nuevo vector.
4. Función update que es el loop main que está en constante ejecución.

5. Función que procesa el mensaje y mueve el objeto, tanto lineal como rotacionalmente.

- Control Coche Entrenamiento:

Para cuando se quiera entrenar la red para conducir por otros terrenos, se ha creado el script de control entrenamiento, que observamos en la Figura 11 y Figura 12. Dicho script se ejecutará si está habilitada la entrada Train mediante la tecla “t”.

La estructura del código es similar al de los anteriores:

1. Función de adquisición de entradas, asigna a las variables las entradas del teclado: “W” hacia adelante, “S” hacia atrás, “A” hacia la izquierda, “D” hacia la derecha, “t” entrenar y “espacio” para frenar.
2. Función que asigna la dirección al objeto.
3. Función para acelerar y frenar entrando en el componente “.motortorque” del objeto para dar fuerza y simular un motor y en el componente “.breaktorque” para aplicar una fuerza de 400.0 N.m para frenar.

```
public void GetInput() // obtengo las entradas para moverme, por defecto están las de
{
    m_horizontalInput = Input.GetAxis("Horizontal");
    m_VerticalInput = Input.GetAxis("Vertical");
    freno = Input.GetButton("Jump");
    entrenar = Input.GetButton("Train");

    if (freno){
        frenar = true;
    }

    else {
        frenar = false;
    }
}

public void Direccion()
{
    m_AnguloDireccion = maxsteerAngle * m_horizontalInput;
    Front_right_wheelW.steerAngle = m_AnguloDireccion;
    Front_left_wheelW.steerAngle= m_AnguloDireccion;
}

public void Aceleracion()
{
    Front_right_wheelW.motorTorque = m_VerticalInput * motorFuerza;
    Front_left_wheelW.motorTorque = m_VerticalInput * motorFuerza;

    if (frenar){
        Front_right_wheelW.brakeTorque = 400.0f;
        Front_left_wheelW.brakeTorque = 400.0f;
    }
    else {
        Front_right_wheelW.brakeTorque = 0.0f;
        Front_left_wheelW.brakeTorque = 0.0f;
    }
}
```

Figura 11, Script Control Coche Funciones.

4. Procedimiento que llama a la función siguiente.
5. Función que cambia la apariencia física y del colisionador de las ruedas según la dirección a la que se vaya a girar.
6. Fixed Update, el main que se estará ejecutando siempre donde están las llamadas a/se llaman todas las funciones.
7. La declaración de variables.

```
private void PosicionRuedas()
{
    PosicionRuedas(Front_right_wheelW,Front_right_wheelT);
    PosicionRuedas(Front_left_wheelW,Front_left_wheelT);
    PosicionRuedas(Back_left_wheelW,Back_left_wheelT);
    PosicionRuedas(Back_right_wheelW,Back_right_wheelT);
}

private void PosicionRuedas(WheelCollider _collider, Transform _transform)
{
    Vector3 _pos = _transform.position;
    Quaternion _quat = _transform.rotation;

    _collider.GetWorldPose(out _pos, out _quat);

    _transform.position = _pos;
    _transform.rotation = _quat;
}

private void FixedUpdate()
{
    GetInput();
    Direccion();
    Aceleracion();
    PosicionRuedas();
}

private float m_horizontalInput;
private float m_VerticalInput;
private float m_AnguloDireccion;
private bool freno;

public WheelCollider Front_right_wheelW, Front_left_wheelW;
public WheelCollider Back_left_wheelW, Back_right_wheelW;
public Transform Front_right_wheelT, Front_left_wheelT;
public Transform Back_right_wheelT, Back_left_wheelT;
public float maxsteerAngle = 30;
public float motorFuerza = 50;
public bool entrenar = false;
```

Figura 12, Script Control Coche Funciones y Declaración de Variables.

Por último, para la configuración del coche es importante asignar al objeto “coche” en los componentes el script de control entrenamiento, asignando también a las entradas del componente los objetos de las ruedas donde corresponda, como vemos en la Figura 13.

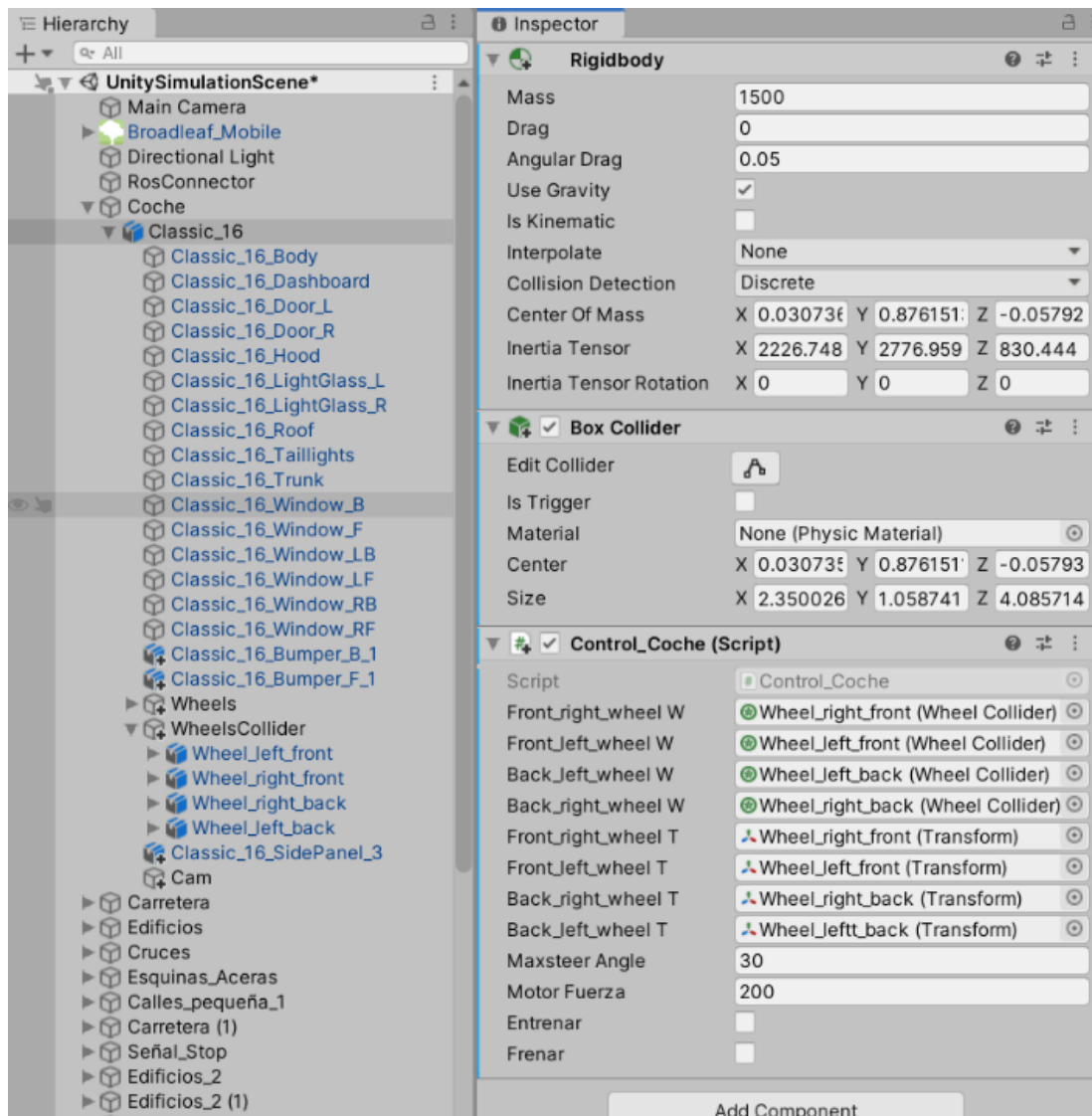


Figura 13, Configuración Componentes Coche.

También hay que añadir los datos en el “Rigidbody” como la masa y decir que es afectado por la gravedad, un “Box Collider” para las colisiones con los demás objetos.

4.3.4 ROS conector

Este objeto es clave para el funcionamiento, puesto que hace de comunicación con el script de control en Python. Hasta ahora se ha explicado más la parte estética del simulador, así como sus accionadores. Pero el objeto *RosConnector* engloba todos los scripts de Ros#, tal y como se aprecia en la Figura 14.

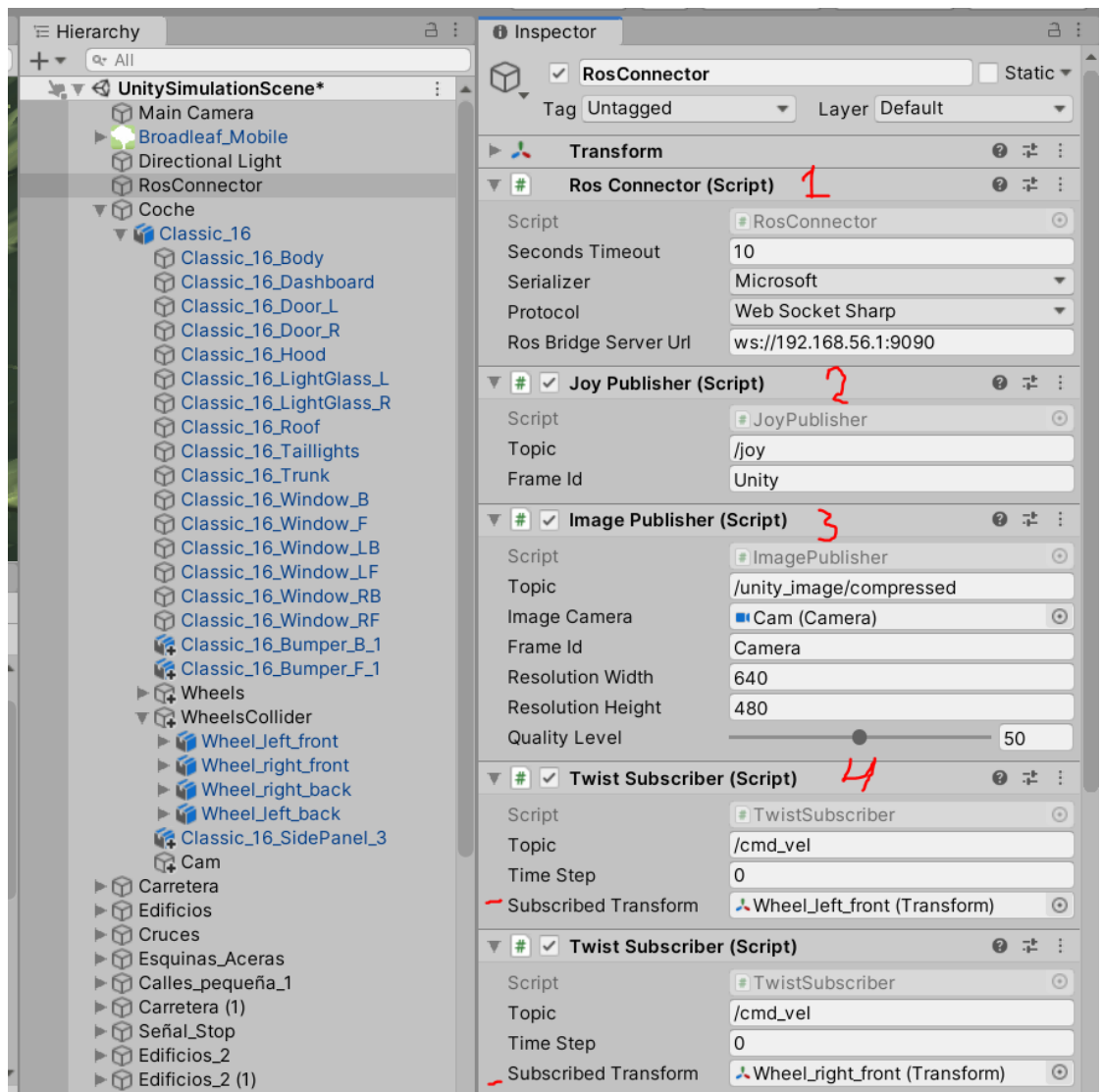


Figura 14, Objeto ROS Conector.

- 1- **RosConnector (Script):** es el encargado de comunicar con el *RosBridge server*, asignándole la *Url* compuesta por la dirección IP de la máquina virtual donde esté instalado *ROS* y *RosBridge*, en este caso 192.168.56.1 y el puerto 9090. En el apartado protocolo asignamos *Web Socket Sharp*.
- 2- **Joy Publisher (Script):** Sólo se utiliza para enviar al script de Python mediante un Button el modo de ejecución: simulación o entrenamiento.
- 3- **Image Publisher (Script):** Es la fuente de adquisición principal de datos que le pasaremos al control en Python, mediante el Topic */unity_image/Compressed*, también se asigna la cámara (nota importante: la cámara asignada es el objeto cámara que se incluyó en el coche, no la cámara principal) y el tamaño de la imagen.
- 4- **Twist Subscriber (Script):** Es el script que hace de accionador para mover el coche en modo simulación, hay dos porque es para diferentes ruedas, aunque sea el mismo script.

5 Bloque 2 – ROS / ROS# y RosBridge

Desde el entorno se obtiene mucha información que tiene que ser enviada al control, ser tratada y luego tomar las decisiones, para finalmente volver al simulador para actuar sobre el vehículo. Ése es el trabajo del Bloque 2, servir de canal entre el “cerebro” y los accionadores.

Primeramente, resulta de interés detallar el proceso de configuración implementado en esta parte. Desde la instalación de la máquina virtual hasta la configuración de los paquetes de ROS.

5.1 Máquina virtual (VM), con sistema operativo Ubuntu 18.04

Se utiliza la máquina virtual que se ejecuta como una aplicación nativa de Windows 10, que es una funcionalidad relativamente nueva. Así evitamos el uso de aplicaciones externas como VMware o Virtualbox que tienden a trabajar más lentamente. En cambio, al ser solo terminal, el rendimiento del subsistema Linux es mucho mejor que las aplicaciones antes mencionadas.

Para instalarlo, primero se tiene que activar la característica “subsistemas de Windows para Linux” en las características de Windows, como vemos en la Figura 15.

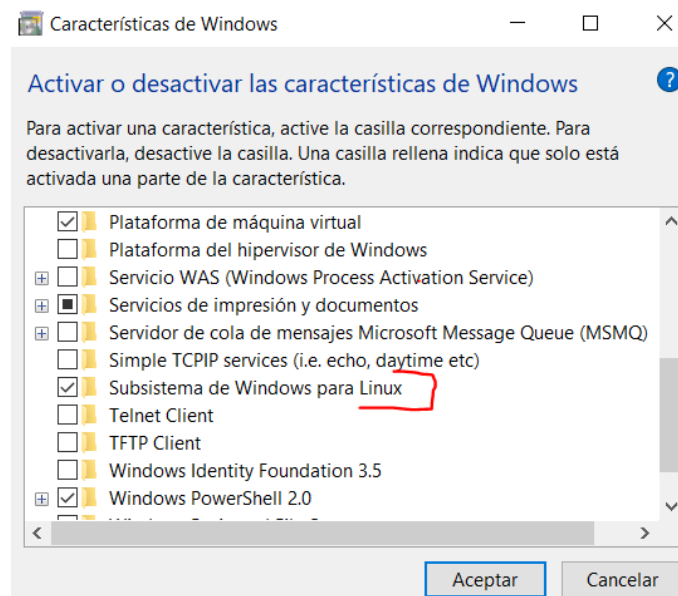


Figura 15, Características de Windows, subsistema para Linux.

Luego se descarga Ubuntu 18.04 LTS desde el Microsoft Store. Tiene que ser esta versión porque la versión de ROS que se instalará será ROS Melodic y únicamente es soportada por la distribución 18.04 (si se va a realizar en la 16.04 entonces se ha de instalar ROS Kinetic, que es la versión previa).

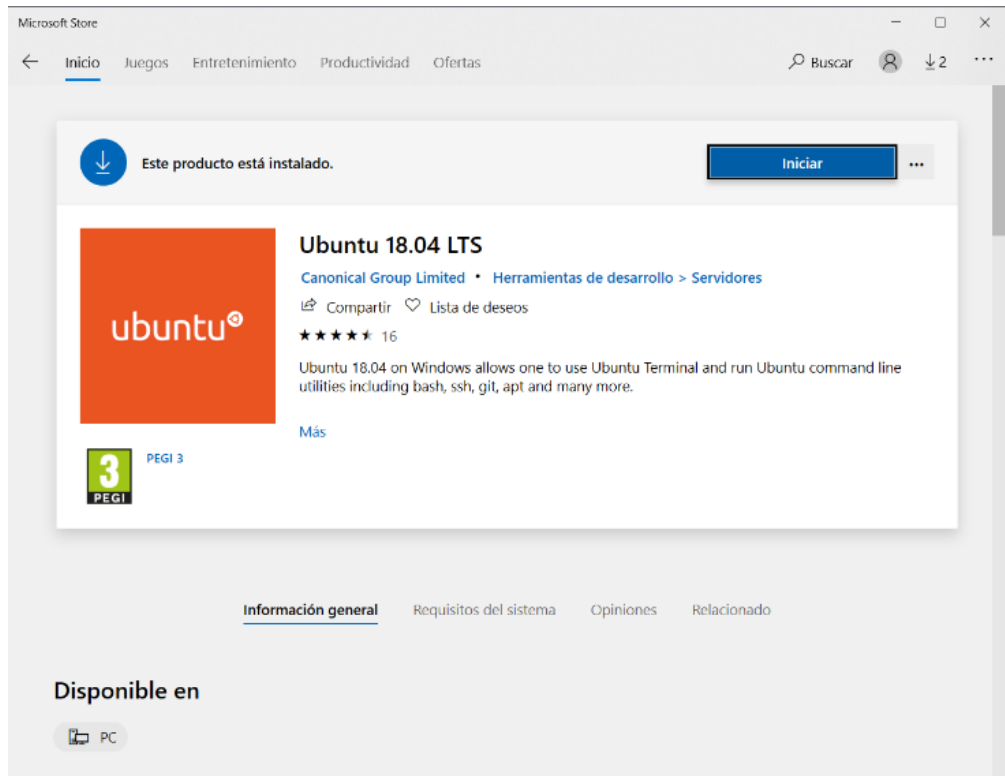


Figura 16, Máquina virtual Ubuntu 18.04 LTS Microsoft Store.

5.2 Instalación de ROS Melodic

Al iniciar la máquina virtual nos pide que configuremos el usuario y la contraseña. Luego nos aparece el terminal que vemos en la Figura 17. Siguiendo los pasos de instalación escribimos los comandos presentes en la Figura 18 y en la Figura 19. Específicamente los que están señalados en rojo.

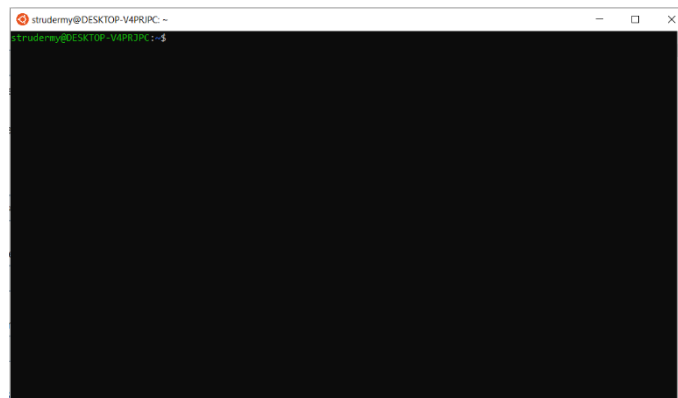


Figura 17, Terminal Ubuntu

1.4 Installation

First, make sure your Debian package index is up-to-date:

```
sudo apt update
```

There are many different libraries and tools in ROS. We provided four default configurations to get you started. You can also install ROS packages individually.

In case of problems with the next step, you can use following repositories instead of the ones mentioned above [ros-shadow-fixed](#)

Desktop-Full Install: (Recommended) : ROS, [rqt](#), [rviz](#), robot-generic libraries, 2D/3D simulators and 2D/3D perception

```
sudo apt install ros-melodic-desktop-full
```

or [click here](#)

Desktop Install: ROS, [rqt](#), [rviz](#), and robot-generic libraries

```
sudo apt install ros-melodic-desktop
```

or [click here](#)

ROS-Base: (Bare Bones) ROS package, build, and communication libraries. No GUI tools.

```
sudo apt install ros-melodic-ros-base
```

or [click here](#)

Individual Package: You can also install a specific ROS package (replace underscores with dashes of the package name):

```
sudo apt install ros-melodic-PACKAGE
```

e.g.

```
sudo apt install ros-melodic-slam-gmapping
```

To find available packages, use:

```
apt search ros-melodic
```

Figura 18, Comandos para instalación ROS Melodic.

1.5 Environment setup

It's convenient if the ROS environment variables are automatically added to your bash session every time a new shell is launched:

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

If you have more than one ROS distribution installed, `~/.bashrc` must only source the `setup.bash` for the version you are currently using.

If you just want to change the environment of your current shell, instead of the above you can type:

```
source /opt/ros/melodic/setup.bash
```

If you use zsh instead of bash you need to run the following commands to set up your shell:

```
echo "source /opt/ros/melodic/setup.zsh" >> ~/.zshrc
source ~/.zshrc
```

1.6 Dependencies for building packages

Up to now you have installed what you need to run the core ROS packages. To create and manage your own ROS workspaces, there are various tools and requirements that are distributed separately. For example, `roscpp` is a frequently used command-line tool that enables you to easily download many source trees for ROS packages with one command.

To install this tool and other dependencies for building ROS packages, run:

```
sudo apt install python-rosdep python-roscpp python-roscpp-generator python-wstool build-essential
```

1.6.1 Initialize rosdep

Before you can use many ROS tools, you will need to initialize `rosdep`. `rosdep` enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS. If you have not yet installed `rosdep`, do so as follows.

```
sudo apt install python-rosdep
```

With the following, you can initialize `rosdep`.

```
sudo rosdep init
rosdep update
```

Figura 19, Configuración Paquetes ROS Melodic.

Después de la instalación, hay que crear el entorno de trabajo e instalar el paquete de RosBridge para la comunicación con Unity.

Procedemos a ejecutar el siguiente comando en la terminal

```
$ source /opt/ros/melodic/setup.bash
```

A continuación, seguimos las siguientes instrucciones de la construcción de un Workspace, como apreciamos en la Figura 20.

Let's create and build a [catkin workspace](#):

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

Figura 20, Comandos para creación catkin workspace.

Por último, ejecutamos el siguiente comando para instalar rosbridge suite:

```
$ sudo apt-get install ros-Melodic-rosbridge-server
```

5.3 Interfaz Gráfico

Como la extensión de Linux en Windows es solo un terminal, es necesario el uso de servidores que permitan visualizar o dar un entorno gráfico, como lo es Xming. Se debe descargar e instalar y luego ejecutar Xlaunch.

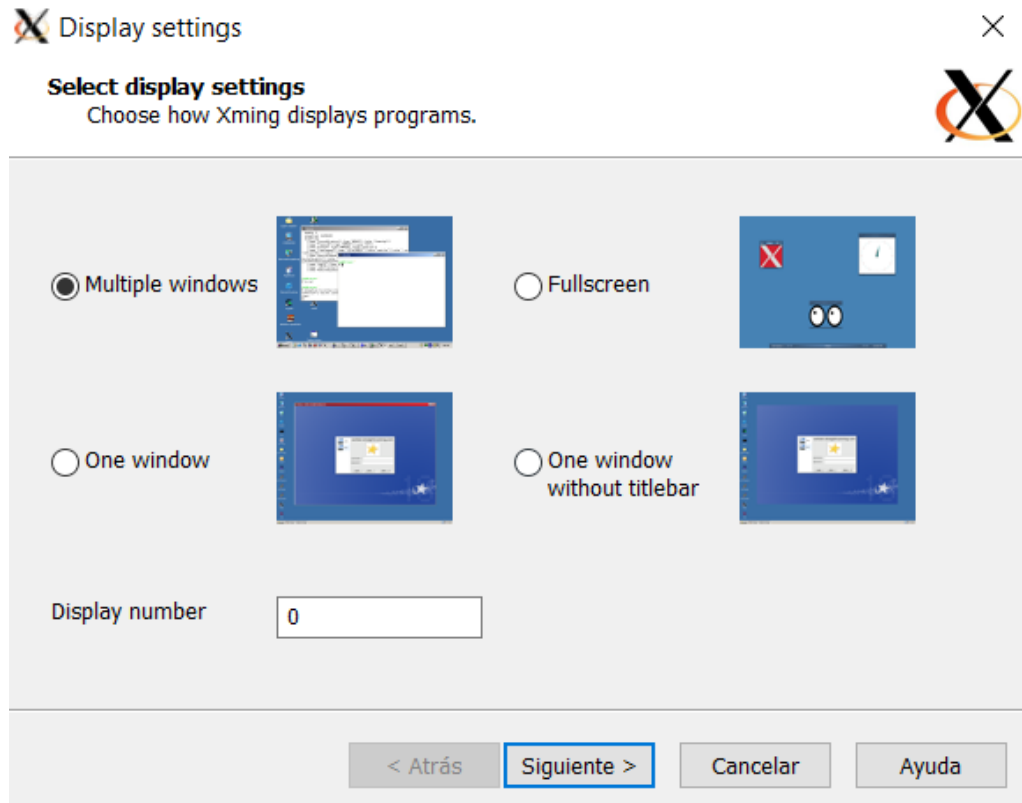


Figura 21, Servidor Xming

Con esto está conectado al servidor Xming. El siguiente paso es instalar en el terminal Ubuntu un administrador de archivos. Elegimos “Nautilus” y así se puede navegar por las carpetas de la máquina virtual.

Para ello hay que ejecutar los siguientes comandos:

```
sudo apt-get update -y
```

```
sudo apt-get install -y nautilus
```

Una vez finalizada la instalación, hay que habilitar el puerto al servidor Xming, con el comando `export DISPLAY=:0.0` (esto sólo la primera vez, luego es posible añadirlo para que se ejecute al abrir la terminal).

5.4 Configuración de Paquetes

Como se sabe ROS es un sistema operativo basado en paquetes, que facilitan el desarrollo de aplicaciones para los robots. Una de las herramientas importantes es la creación de paquetes, que bien se pueden crear y personalizar según la necesidad del usuario o importarlos de repositorios online.

5.4.1 Configuración del Paquete TFG-Scripts

Se ha creado un paquete llamado tfg-control-script que aloja al script de control en Python. Además, alberga el modelo de RNA y los datos recolectados por la cámara del simulador. Para ello hay que realizar una serie de pasos ejecutables vía la terminal de comandos, los cuales se explican a continuación:

- 1- **cd ~/catkin_ws/src:** para acceder a la carpeta src del Working Space que se ha creado anteriormente.
- 2- **catkin_create_pkg <tfg-control-script> [roscpp] [rospy] [std_msgs][cv_bridge]:** para crear el paquete con las dependencias correspondientes.
- 3- **cd ~/catkin_ws:** volvemos a la carpeta anterior.
- 4- **catkin_make:** para añadir los archivos CMake al flujo de trabajo.
- 5- **/catkin_ws/devel/setup.bash:** añade el espacio de trabajo al entorno ROS.

Posteriormente, dentro de la carpeta src del paquete tfg-control-script, se crea el script de control que en este caso se llama main.py, tal y como se observa en la Figura 22.

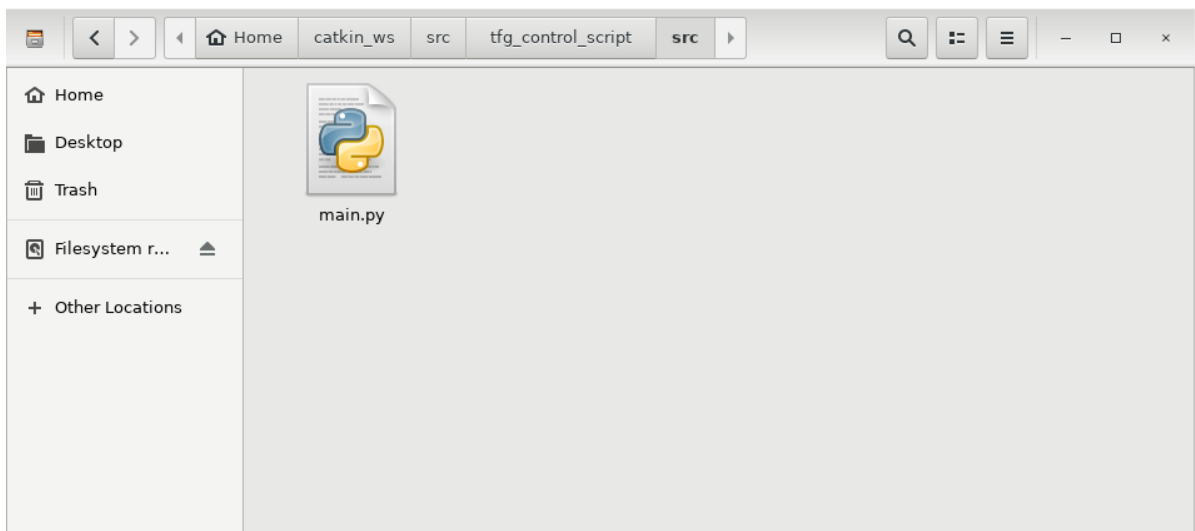


Figura 22, Vista del script dentro del paquete creado

5.4.2 Configuración de los Paquetes de Ros#

El paso anterior es para agregar un paquete desde cero, pero los paquetes de ROS# ya están creados, lo que no están es añadidos al entorno de trabajo. Para añadirlos basta con descargar

de internet (desde el enlace: <https://github.com/siemens/ros-sharp/tree/master/ROS>) los siguientes paquetes y colocarlos dentro de la carpeta src del espacio de trabajo catkin_ws:

- 1- FileServer
- 2- Unity_simulation_scene

Luego hay que ejecutar los comandos que se muestran a continuación:

- `cd ~/catkin_ws.`
- `catkin_make.`
- `/catkin_ws/devel/setup.bash.`

Una vez hecho esto, ya se dispone de las principales herramientas para comunicar, por supuesto a falta de la información.

5.5 Comunicación entre entorno Unity y control en Python.

Como bien se ha mencionado, este bloque es el que permite el intercambio de información entre el bloque de comandos (Python) y el bloque de acción (Unity), para ello se aprovechan las herramientas que dispone ROS para establecer comunicaciones (nodos, Topics y mensajes).

5.5.1 Topics Utilizados

Los Topics son los canales a los cuales los nodos se suscribirán para enviar o recibir información que contienen los mensajes. Los nombres de los Topics se asignan de manera que tengan relación con el mensaje, pero es importante que el tópico sea el mismo en ambos nodos porque sino no llega a comunicar, básicamente porque se estaría enviando/recibiendo información a Topics diferentes.

Los Topics que se utilizan en este proyecto son:

- `/Joy`
- `/unity_image/compressed`
- `/cmd-vel`

5.5.2 Mensajes Utilizados

Los mensajes son la estructura de datos en la cual se enviará la información, ROS permite crear tus propios mensajes, pero en este caso se utilizaron los mensajes estándar de ROS.

- **sensor_msgs/Joy.msg:** describe el estado de los ejes y botones de un joystick. Está compuesto de:
 - Header “header”.
 - Float32[] “axes”, para los ejes.
 - Int32[] “buttons”, para los botones.

- **sensor_msgs/CompressedImage.msg:** este mensaje contiene una imagen comprimida. Está compuesto por:
 - Header “header”.
 - String “format” (especifica el formato de la imagen jpg, png).
 - UInt8[] “data” es el buffer de la imagen comprimida.
- **geometry_msgs/Twist.msg:** expresa la velocidad de un objeto, indicando sus componentes lineales y angulares en el espacio. Está compuesto de:
 - Vector3 linear.
 - Vector3 angular.

5.5.3 Scripts de Comunicación por ROS# (Nodos)

Estos son los nodos que están incluidos en el paquete de asset de ROS# instalado en el Unity, pero resulta conveniente explicarlo en este apartado, ya que en el Unity solo hay que añadir los objetos a los campos correspondientes. A continuación, se describe el código y alguna modificación que se ha hecho.

Para enviar datos desde el Unity, se utilizan los scripts “Publisher” (publicador/editor), que publican la información en los Topics (/Unity/ImageCompressed y /Joy). Luego, en el script de control (Python) se suscriben a estos para utilizar la información en los mensajes. Por otro lado, información que devuelve la red publicando la información en el tópico (/cmd_vel) en los scripts “Subscriber” (suscriptor).

5.5.3.1 *Nodo Subscriber*

- **Twist Subscriber:**

En la Figura 23 está estructurado todo el código del nodo Twist Subscriber, el cual está programado en C# y dentro del asset ROS# de Unity. Es importante ver que al inicio se declara un ámbito llamado “RosSharp.RosbridgeClient”, esto es para poder asignar nombres únicos a las clases que harán referencia a los nodos, así como también trabajar en el mismo espacio y poder utilizar las características de ROS#.

También es importante destacar que, al declarar la clase, se debe definir que es un Subscriber y decirle qué tipo de mensaje espera (Unity Subscriber <MessageTypes.Geometry.Twist>).

La estructura del script es la siguiente:

```

using UnityEngine;

namespace RosSharp.RosBridgeClient
{
    public class TwistSubscriber : UnitySubscriber<MessageTypes.Geometry.Twist>
    {
        public Transform SubscribedTransform;

        private float previousRealTime;
        private Vector3 linearVelocity;
        private Vector3 angularVelocity;
        private bool isMessageReceived;

        protected override void Start()
        {
            base.Start();
        }

        protected override void ReceiveMessage(MessageTypes.Geometry.Twist message)
        {
            linearVelocity = ToVector3(message.linear).Ros2Unity();
            angularVelocity = -ToVector3(message.angular).Ros2Unity();
            isMessageReceived = true;
        }

        private static Vector3 ToVector3(MessageTypes.Geometry.Vector3 geometryVector3)
        {
            return new Vector3((float)geometryVector3.x, (float)geometryVector3.y, (float)geometryVector3.z);
        }

        private void Update()
        {
            if (isMessageReceived)
                ProcessMessage();
        }

        private void ProcessMessage()
        {
            float deltaTime = Time.realtimeSinceStartup - previousRealTime;

            SubscribedTransform.Translate(linearVelocity * deltaTime);
            SubscribedTransform.Rotate(Vector3.forward, angularVelocity.x * deltaTime);
            SubscribedTransform.Rotate(Vector3.up, angularVelocity.y * deltaTime);
            SubscribedTransform.Rotate(Vector3.left, angularVelocity.z * deltaTime);

            previousRealTime = Time.realtimeSinceStartup;

            isMessageReceived = false;
        }
    }
}

```

Figura 23, Script Twist Subscriber.

- 1- **Declaración de variables.**
- 2- **Función de mensaje recibido:** cuando recibe un mensaje del tipo Twist desde el Topic que le indicamos desde fuera, convierte en Vector3 el mensaje recibido desde ROS.
- 3- **Función “ToVector3”:** genera un vector nuevo tomando las componentes x, y, z del mensaje que se recibe.
- 4- **Función update:** actualiza si ha recibido un mensaje.
- 5- **Función procesar mensaje:** es donde realiza las acciones al objeto real en Unity, en este caso a las ruedas le aplica la velocidad lineal y rotacional dependiendo del mensaje recibido.

5.5.3.2 Nodos Publisher

5.5.3.3

- **Image Publisher:**

De igual forma que en el Subscriber se declara un ámbito llamado “RosSharp.RosbridgeClient”. Al declarar la clase, se debe definir que es un Publisher y decirle qué tipo de mensaje publica (Unity Publisher<MessageTypes.Sensor.CompressedImage>).

```
namespace RosSharp.RosBridgeClient
{
    public class ImagePublisher : UnityPublisher<MessageTypes.Sensor.CompressedImage>
    {
        public Camera ImageCamera;
        public string FrameId = "Camera";
        public int resolutionWidth = 640;
        public int resolutionHeight = 480;
        [Range(0, 100)]
        public int qualityLevel = 50;

        private MessageTypes.Sensor.CompressedImage message;
        private Texture2D texture2D;
        private Rect rect;

        protected override void Start()
        {
            base.Start();
            InitializeGameObject();
            InitializeMessage();
            Camera.onPostRender += UpdateImage;
        }

        private void UpdateImage(Camera _camera)
        {
            if (texture2D != null && _camera == this.ImageCamera)
                UpdateMessage();
        }

        private void InitializeGameObject()
        {
            texture2D = new Texture2D(resolutionWidth, resolutionHeight, TextureFormat.RGB24, false);
            rect = new Rect(0, 0, resolutionWidth, resolutionHeight);
            ImageCamera.targetTexture = new RenderTexture(resolutionWidth, resolutionHeight, 24);
        }

        private void InitializeMessage()
        {
            message = new MessageTypes.Sensor.CompressedImage();
            message.header.frame_id = FrameId;
            message.format = "jpeg";
        }

        private void UpdateMessage()
        {
            message.header.Update();
            texture2D.ReadPixels(rect, 0, 0);
            message.data = texture2D.EncodeToJPG(qualityLevel);
            Publish(message);
        }
    }
}
```

Figura 24, Script Image Publisher.

En la Figura 24 se observa la estructura del nodo que se divide en:

- 1- Declaración de variables.
- 2- Inicialización de objetos y mensaje.
- 3- Actualización del mensaje cuando hay un cambio en la imagen.
- 4- Inicialización del objeto en este caso la cámara, es donde se le asigna la resolución, el tamaño, el formato de colores.
- 5- Inicialización del mensaje, indicando el header y el formato de la imagen.

6- Función que actualiza el mensaje codificando en message.data a jpg.

- **Joy Publisher Mine**

El encabezado es igual que el Publisher anterior, salvo el tipo de mensaje. Cabe destacar que este script en particular fue modificado por ello se llama JoyPublisherMine, porque era necesario obtener las entradas desde el teclado y no desde un JoyReader como estaba planeado el script original. Se ha añadido la función “GetInputs()” y todo lo relacionado con ella (variables), además se ha modificado la función update message para que escribiera en axes los valores de 1 y -1 en los ejes 0 y 1 que indican movimientos horizontales (izquierda, derecha) y verticales (arriba, abajo) respectivamente, mientras que en buttons escribe un 1 si se está entrenando o un 0 si se está simulando.

En la Figura 25 se observa el script original y en la Figura 26 inicia el script terminando en la Figura 27.

```
namespace RosSharp.RosBridgeClient
{
    public class JoyPublisher : UnityPublisher<MessageTypes.Sensor.Joy>
    {
        private JoyAxisReader[] JoyAxisReaders;
        private JoyButtonReader[] JoyButtonReaders;

        public string FrameId = "Unity";

        private MessageTypes.Sensor.Joy message;

        protected override void Start()
        {
            base.Start();
            InitializeGameObject();
            InitializeMessage();
        }

        private void Update()
        {
            UpdateMessage();
        }

        private void InitializeGameObject()
        {
            JoyAxisReaders = GetComponents<JoyAxisReader>();
            JoyButtonReaders = GetComponents<JoyButtonReader>();
        }

        private void InitializeMessage()
        {
            message = new MessageTypes.Sensor.Joy();
            message.header.frame_id = FrameId;
            message.axes = new float[JoyAxisReaders.Length];
            message.buttons = new int[JoyButtonReaders.Length];
        }

        private void UpdateMessage()
        {
            message.header.Update();

            for (int i = 0; i < JoyAxisReaders.Length; i++)
                message.axes[i] = JoyAxisReaders[i].Read();

            for (int i = 0; i < JoyButtonReaders.Length; i++)
                message.buttons[i] = (JoyButtonReaders[i].Read() ? 1 : 0);

            Publish(message);
        }
    }
}
```

Figura 25, Script Joy Publisher Original.

```

using UnityEngine;
using System.Collections;

namespace RosSharp.RosBridgeClient
{
    public class JoyPublisherMine : UnityPublisher<MessageTypes.Sensor.Joy>
    {
        private JoyButtonReader[] JoyButtonReaders;
        private bool estado_actual;
        private bool estado_anterior = false;
        public bool entrenando;
        private bool train;
        public string FrameId = "Unity";
        private float m_horizontalInput;
        private float m_VerticalInput;
        private bool freno;

        private MessageTypes.Sensor.Joy message;

        protected override void Start()
        {
            base.Start();
            InitializeGameObject();
            InitializeMessage();
        }

        private void GetInputs(){

            train = Input.GetButton("Train") ;
            m_horizontalInput = Input.GetAxis("Horizontal");
            m_VerticalInput = Input.GetAxis("Vertical");
            freno = Input.GetButton("Jump");
        }

        private void Update()
        {
            UpdateMessage();
            GetInputs();
        }

        private void InitializeGameObject()
        {
            JoyButtonReaders = GetComponents<JoyButtonReader>();
        }

        private void InitializeMessage()
        {
            message = new MessageTypes.Sensor.Joy();
            message.header.frame_id = FrameId;
            message.axes = new float[2];
            message.buttons = new int[2];
        }
    }
}

```

Figura 26, Script Joy Publisher Mine parte I.

La estructura del programa viene dada por:

- 1- Declaración de variables.
- 2- Llamada a la inicialización de objeto y mensaje.
- 3- Función para obtener los datos desde el teclado.
- 4- Función “update”.
- 5- Inicialización del mensaje de tipo Joy.
- 6- Actualización del mensaje.

```

private void UpdateMessage()
{
    message.header.Update();

    message.axes[0]=m_VerticalInput;
    message.axes[1]=m_horizontalInput;

    estado_actual = train;

    if (estado_actual == true && estado_anterior == false )
    {
        entrenando = !entrenando;
    }
    estado_anterior = estado_actual;

    if (entrenando){
        message.buttons[0] = 1;
    }
    else message.buttons[0] = 0;

    if (freno){
        message.buttons[1] = 1;
    }

    else {
        message.buttons[1] = 0;
    }

    Publish(message);
}
}

```

Figura 27, Script Joy Publisher Mine parte II.

5.6 Red de comunicación

En la Figura 28 aparecen los nodos activos en óvalo, en rectángulo los Topics y las flechas indican el flujo, la punta de flecha Subscriber y la cola de flecha Publisher, lo cual corresponde a lo que se ha explicado con anterioridad. Por otro lado, cuando se está en simulación se aprecia en la Figura 29 el flujo de información.

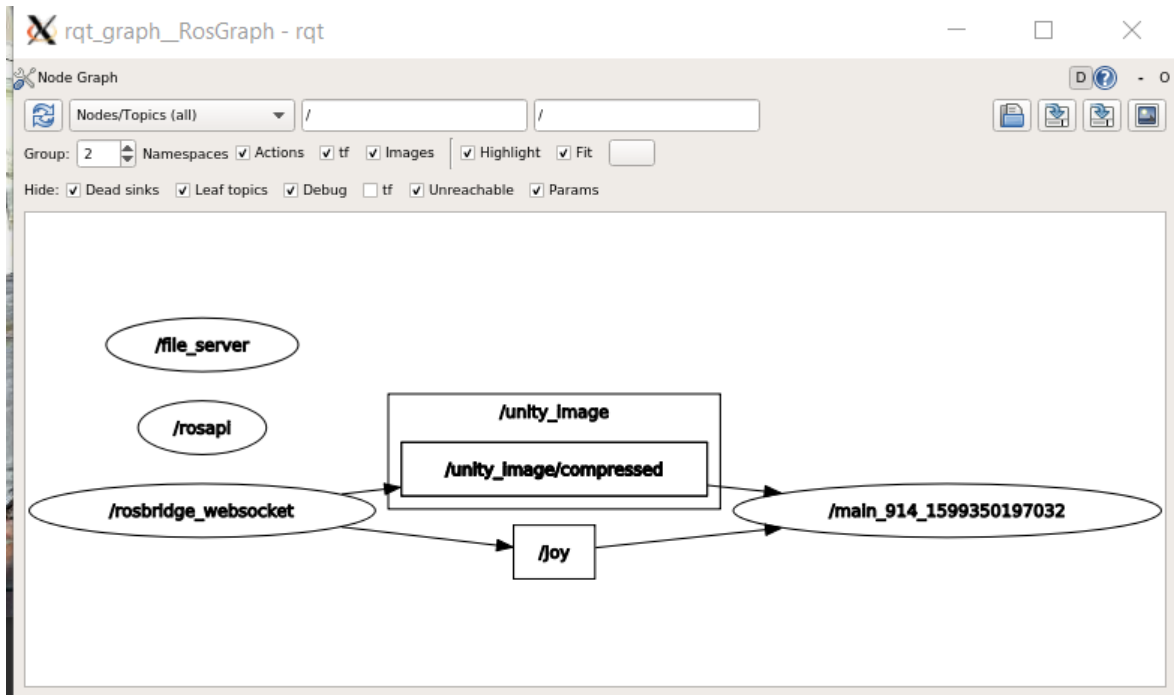


Figura 28, Red de flujo de información entre nodos y Topics (modo entrenamiento).

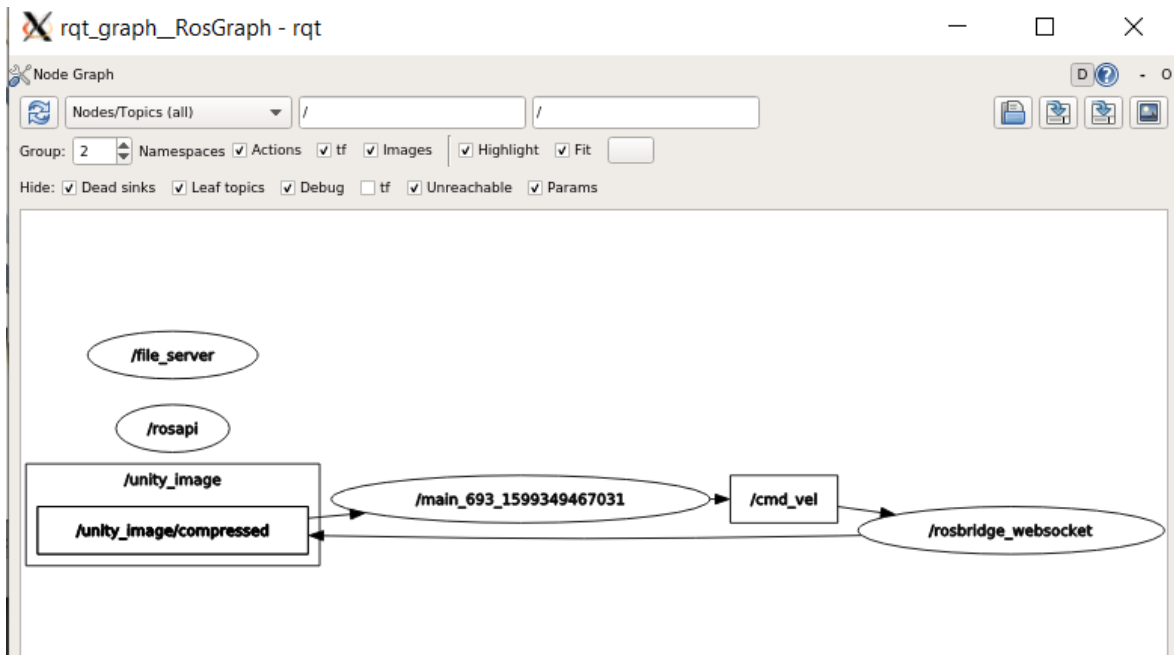


Figura 29, Red de flujo de información entre nodos y Topics (modo simulación).

6 Bloque 3 – Script de control en Python y Red neuronal

Por último, se ha desarrollado un script a partir de uno que ya estaba en un proyecto en internet sobre la conducción autónoma para un coche a control remoto. Este tiene asociado una cámara mediante la cual hace el procesado de la información, enviando los datos del “stream” a una tarjeta Raspberry PI y también posee un Arduino para ejecutar las acciones que le ordena la RNL . Al proyecto se puede acceder desde el siguiente enlace: <https://github.com/hamuchiwa/AutoRCCar>.

Se ha eliminado la mayoría, salvando la red neuronal y la manera de entrenarla, pero la forma de adquirir los datos y la toma de decisiones se ha programado desde cero, puesto que el proyecto “AutoRCCar” recibía las imágenes desde una cámara, las recibe una raspberryPI y ésta envía al ordenador por un host. Mientras que las órdenes al coche se las envía mediante un puerto serie a un Arduino. Todo esto se elimina y se trabaja directamente con ROS.

Para explicarlo de una mejor manera, se divide en 4 segmentos el código, explicando detalladamente cada uno de ellos.

6.1 Segmento 1: *RCDriving*

Es el que se ejecuta cuando se está trabajando en modo simulación, está compuesto de 2 clases.

La primera clase que se define es para el control de la dirección y posee 2 métodos. Los cuales se pueden apreciar en la Figura 30.


```

class RCControl(object):
    def __init__(self):
        self.pub_twist = rospy.Publisher('cmd_vel', Twist, queue_size = 10)
        self.vel_msg = Twist()

    def steer(self, prediction):
        if prediction == 2:
            self.vel_msg.linear.z = 0.0
            self.vel_msg.linear.x = 2
            self.vel_msg.linear.y = 0.0
            self.vel_msg.angular.x = 0
            self.vel_msg.angular.y = 0
            self.vel_msg.angular.z = 0
            print("Forward")
        elif prediction == 0:
            self.vel_msg.angular.x = 1
            self.vel_msg.angular.y = 0
            self.vel_msg.angular.z = -2
            print("Left")
        elif prediction == 1:
            self.vel_msg.angular.x = 1
            self.vel_msg.angular.y = 0
            self.vel_msg.angular.z = 2
            print("Right")
        else:
            self.vel_msg.linear.x = 0
            self.pub_twist.publish(self.vel_msg)

```

Figura 30, Clase RCControl.

Es importante destacar que cuando se crea un método hay que pasarle como dato “self” que es él mismo y así se puede utilizar una variable de un método dentro de otro método ya que está instanciado.

- El método `__init__` que se ejecuta cada vez que se cree un objeto de clase “RCcontrol”, allí es donde se define el Topic a donde va a publicar mediante la línea de comando con la estructura `rospy.publisher(‘nombre tópico’, tipo de mensaje, tiempo de cola)` y se asigna a una variable que sea de tipo mensaje Twist.
- Por otro lado, el método `steer` hay que pasarle como argumento la predicción que realiza la red neuronal.

La segunda clase se llama `RCDriverNNOnly`, es la que se encarga de recibir la imagen y pasársela a la red neuronal para que haga la predicción. Está compuesta de 3 métodos.

```

class RCDriverNNOnly(object):

    def __init__(self, model_path):

        self.sub_image = rospy.Subscriber("/unity_image/compressed", CompressedImage, self.image_callback)
        self.sub_joy = rospy.Subscriber("/Joy", Joy, self.joy_callback)
        rospy.init_node('main', anonymous=True)
        # load trained neural network
        self.nn = NeuralNetwork()
        self.nn.load_model(model_path)
        self.rc_car = RCControl()

    def joy_callback(self, data):
        self.axes = data.axes
        self.buttons = data.buttons

    def image_callback(self, data):
        self.imagen = CvBridge().compressed_imgmsg_to_cv2(data, desired_encoding='passthrough')

    def drive(self):
        global aux
        image = self.imagen
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        b = self.axes

        # lower half of the image
        height, width = gray.shape
        roi = gray[int(height/2):height, :]

        cv2.imshow('image', image)
        # cv2.imshow('mlp_image', roi)

        # reshape image
        image_array = roi.reshape(1, int(height/2) * width).astype(np.float32)

        # neural network makes prediction
        prediction = self.nn.predict(image_array)
        self.rc_car.steer(prediction)

        if b[0]==0:
            aux = 0

```

Figura 31, Clase RCDriverNNOnly.

En Figura 31, se observa la estructura:

- Método `__init__` se suscribe al Topic que se le indica mediante la instrucción `rospy.Subscriber("nombre del Topic", tipo de mensaje, función callback)`, la función callback (en este caso es un método) se llama para asignar a una variable la información que está recibiendo. También en este método se inicia el nodo "main".
- Método `image_callback`, es la función *callback* al que se le pasa como argumento `data`, que ya se le pasa cuando se suscribe al Topic y se llama a la función callback en el método anterior. Asignando a la variable `self.imagen` el valor de `data` y mediante el operador "`CvBridge.compressed_imgmsg_to_cv2`" convertir el tipo de dato a uno que se pueda manejar para procesar la información.
- Método `joy_callback` es la función *callback* para el mensaje tipo Joy, asigna los valores de los ejes y los botones a las variables `self.axes` y `self.buttons`.
- Método `drive` es donde todo converge, se procesa la imagen, se convierte en un array y luego se le pasa a la red neuronal para que haga una predicción y por último se recibe la predicción y se le pasa al método `steer` la clase anterior para que publique el mensaje tipo Twist correspondiente y si el valor de `button[0]` es 0 que corresponde a la tecla "t" se activa al modo "train".

6.2 Segmento 2: *Collect data training*

Este es el segmento más extenso, puesto que está conformado por 2 clases y 2 funciones, lo cual tiene sentido ya que se encarga de recolectar datos, luego los guarda en un archivo .npz y se los transfiere a la red para que los procese.

La primera clase que se define es la clase *NeuralNetwork* que no se ha modificado en prácticamente nada, sólo en el directorio donde se guardarán los archivos. Es importante crear dichas carpetas en los directorios con el explorador de archivos Nautilus, de lo contrario no creará nada.

```
class NeuralNetwork(object):
    def __init__(self):
        self.model = None

    def create(self, layer_sizes):
        # create neural network
        self.model = cv2.ml.ANN_MLP_create()
        self.model.setLayerSizes(np.int32(layer_sizes))
        self.model.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP)
        self.model.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM, 2, 1)
        self.model.setTermCriteria((cv2.TERM_CRITERIA_COUNT, 100, 0.01))

    def train(self, X, y):
        # set start time
        start = time.time()

        print("Training ...")
        self.model.train(np.float32(X), cv2.ml.ROW_SAMPLE, np.float32(y))

        # set end time
        end = time.time()
        print("Training duration: %.2fs" % (end - start))

    def evaluate(self, X, y):
        ret, resp = self.model.predict(X)
        prediction = resp.argmax(-1)
        true_labels = y.argmax(-1)
        accuracy = np.mean(prediction == true_labels)
        return accuracy

    def save_model(self, path):
        directory = "/home/strudermey/catkin_ws/src/tfg_control_script/Model"
        if not os.path.exists(directory):
            os.makedirs(directory)
        self.model.save(path)
        print("Model saved to: " + "" + path + "")

    def load_model(self, path):
        if not os.path.exists(path):
            print("Model does not exist, exit")
            sys.exit()
        self.model = cv2.ml.ANN_MLP_load(path)

    def predict(self, X):
        resp = None
        try:
            ret, resp = self.model.predict(X)
        except Exception as e:
            print(e)
        return resp.argmax(-1)
```

Figura 32, Clase Red Neuronal.

En la Figura 32 se observa la estructura con todos sus métodos:

- **Init:** crea la variable model.
- **Create:** como argumento se le pasa el tamaño de las capas, en este método se asigna el tipo de red neuronal en este caso perceptrón multicapa (MLP), se le asigna el algoritmo de aprendizaje “backpropagation”, la función de activación que es la

sigmoidal simétrica y, por último, el criterio de iteración que se ha definido una cuenta hasta 100.

- **Train:** se le pasa como argumento dos vectores el vector X que es la entrada y el vector y que es la salida que se espera que de la red para dicha entrada.
- **Evaluate:** evalúa el vector X y devuelve un valor en el vector y .
- **Save_model:** guarda el modelo en la dirección que se le indica.
- **Load_model:** carga el modelo en la dirección que se le indica en el argumento *path*.
- **Predict:** realiza una predicción con la imagen que se le envía en el vector X .

La segunda clase que se define es CollectDataTraining, utilizada para guardar datos en archivos .npz. En la Figura 33 se puede ver la primera parte.

```
class CollectDataTraining(object):
    def __init__(self, input_size):
        self.input_size = input_size
        self.sub_image = rospy.Subscriber("/unity_image/compressed", CompressedImage, self.image_callback)
        self.sub_joy = rospy.Subscriber("/Joy", Joy, self.joy_callback)
        self.saved_frame = 0
        self.total_frame = 0
        self.X = np.empty((0, self.input_size))
        self.y = np.empty((0, 4))
        rospy.init_node('main', anonymous=True)
        self.k = np.zeros((4, 4), 'float')
        for i in range(4):
            self.k[i, i] = 1
    def image_callback(self, data):
        self.imagen = CvBridge().compressed_imgmsg_to_cv2(data, desired_encoding='passthrough')
    def joy_callback(self, data):
        self.axes = data.axes
        self.buttons = data.buttons
    def collect(self):
        global aux
        image = self.imagen
        a = self.axes
        b = self.buttons

        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        height, width = gray.shape
        roi = gray[int(height/2):height, :]
        temp_array = roi.reshape(1, int(height/2) * width).astype(np.float32)
        cv2.imshow('image', image)
        cv2.waitKey(1)
        #frame += 1
        self.total_frame += 1
```

Figura 33, Clase CollectDataTraining parte I

Compuesta por los métodos:

- **Init:** se le pasa como argumento el tamaño de la imagen (que tiene que ser la mitad de la cámara), dentro se suscribe a los Topics /unity_image/compressed y a /joy, inicializa los vectores X e y como vectores vacíos con un tamaño de 4 para el vector y y del tamaño de la imagen para el vector X , se inicia el nodo *main* y, por último, se crea una matriz auxiliar de tamaño 4x4 de valor unitario para todos sus campos .
- **Image_callback:** función *callback* para la imagen, igual que en el segmento anterior.
- **Joy_callback:** función *callback* para el mensaje tipo Joy, asigna los valores de los ejes y los botones a las variables *self.axes* y *self.buttons*.
- **Collect:** asigna a variables los valores de los *callback*, se procesa la imagen se convierte en un array, luego como se observa en la Figura 34. Consulta los valores que se reciben en los ejes del mensaje tipo Joy y según la condición realiza una

acción, la cual consiste en guardar en el vector *X* apilando verticalmente el valor que tiene actualmente junto al array que tenga el buffer de la imagen procesada, de igual forma lo hace con el vector *y*, pero lo hace con el valor que tiene actualmente y el valor que tenga la matriz auxiliar. Por último, si se desactiva el modo “train”, sale de la recolección y pasa a entrenar.

```

# get input from human driver

# complex orders
if a[0]==1 and a[1]==1:
    print("Forward Right")
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[1]))
    self.saved_frame += 1

elif a[0]==1 and a[1]==-1:
    print("Forward Left")
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[0]))
    self.saved_frame += 1

elif a[0]==-1 and a[1]==1:
    print("Reverse Right")

elif a[0]==-1 and a[1]==-1:
    print("Reverse Left")

# simple orders
elif a[0]==1:
    print("Forward")
    self.saved_frame += 1
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[2]))

elif a[0]==-1:
    print("Reverse")

elif a[1]==1:
    print("Right")
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[1]))
    self.saved_frame += 1

elif a[1]==-1:
    print("Left")
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[0]))
    self.saved_frame += 1

elif b[0]==1: #guardar datos y salir
    aux = False

```

Figura 34, Clase *CollectDataTraining* parte II.

La primera función de este segmento se llama *Load_data* y se utiliza para cargar los datos de un archivo *.npz* y dárselos a la red neuronal para que empiece a entrenar, tal y como se puede observar en la Figura 35. Busca el archivo en el directorio y le pasa los datos que se llamen *train* y *train_labels*.

```

def load_data(input_size, path):
    print("Loading training data...")
    start = time.time()

    # load training data
    X = np.empty((0, input_size))
    y = np.empty((0, 4))
    training_data = glob.glob(path)

    # if no data, exit
    if not training_data:
        print("Data not found, exit")
        sys.exit()

    for single_npz in training_data:
        with np.load(single_npz) as data:
            train = data['train']
            train_labels = data['train_labels']
            X = np.vstack((X, train))
            y = np.vstack((y, train_labels))

        print("Image array shape: ", X.shape)
        print("Label array shape: ", y.shape)

    end = time.time()
    print("Loading data duration: %.2fs" % (end - start))

    # normalize data
    X = X / 255.

    # train validation split, 7:3
    return train_test_split(X, y, test_size=0.3)

```

Figura 35, Función `load_data`.

La segunda función es la de entrenar, carga los datos en las variables y se las pasa a la RNA, llama al método `train` que pertenece a la clase `red neuronal` y empieza a entrenar. Al finalizar guarda los datos en la ruta `model_path` que se observa en la Figura 36.

```

def entrenar():
    input_size = 480 * 320
    data_path = "/home/strudermy/catkin_ws/src/tfg_control_script/Model/data_collect/*.npz"

    X_train, X_valid, y_train, y_valid = load_data(input_size, data_path)

    # train a neural network
    layer_sizes = [input_size, 32, 4]
    nn = NeuralNetwork()
    nn.create(layer_sizes)
    nn.train(X_train, y_train)

    # evaluate on train data
    train_accuracy = nn.evaluate(X_train, y_train)
    print("Train accuracy: ", "{0:.2f}%".format(train_accuracy * 100))

    # evaluate on validation data
    validation_accuracy = nn.evaluate(X_valid, y_valid)
    print("Validation accuracy: ", "{0:.2f}%".format(validation_accuracy * 100))

    # save model
    model_path = "/home/strudermy/catkin_ws/src/tfg_control_script/Model/save_model/nn_model.xml"
    nn.save_model(model_path)

```

Figura 36, Función `entrenar`

6.3 Segmento 3: función *main*

Es el programa principal donde se hace el llamado a las funciones y se crean objetos de las clases explicadas en los segmentos anteriores. Es relativamente corto, lo que se hace es consultar el valor de la variable global “aux”, que dentro de cada clase adquiere valor 0 dependiendo del valor del botón [0] correspondiente a la tecla “t” que activa o desactiva el modo entrenamiento.

```
if __name__ == '__main__':
    s = 480 * 320
    mp= "/home/strudermy/catkin_ws/src/tfg_control_script/Model/save_model/nn_model.xml"
    while True:
        ctd=CollectDataTraining(s)
        if aux:
            while aux:
                ctd.collect()
                entrenar()
            else:
                car=RCDriverNNOnly(mp)
                while not aux:
                    car.drive()
```

Figura 37, Programa principal.

En la Figura 37, se observa la estructura y de cómo está dentro de un bucle *while true* para que se ejecute siempre a menos que se cierre el sistema.

6.4 Segmento 4: encabezado

Este último segmento es importante puesto que es donde se incluyen todas las librerías que son utilizadas por el script, en la Figura 38 se observan todas. Además, se importa el método de aprendizaje del modelo `sklearn.model_selection`. También es importante la primera línea ya que indica que es un entorno de Python.

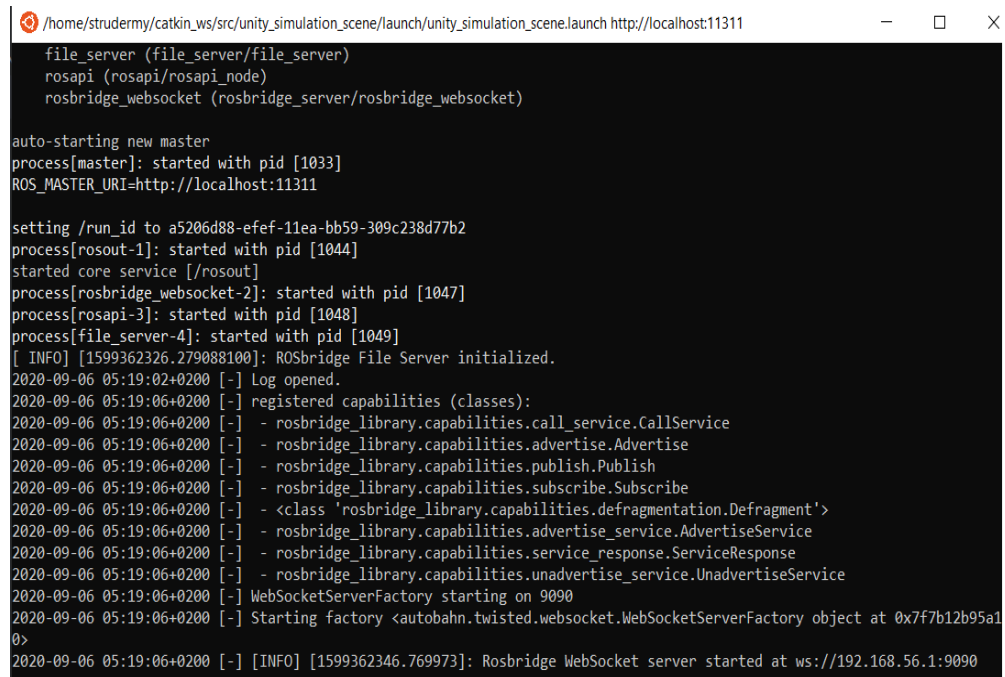
```
#!/usr/bin/env python
import numpy as np
import cv2
import rospkg
import time
import os
import rospy
import glob
import roslib
from sklearn.model_selection import train_test_split
from cv_bridge import CvBridge
from sensor_msgs.msg import CompressedImage
from sensor_msgs.msg import Joy
from geometry_msgs.msg import Twist
aux = True
```

Figura 38, Encabezado inclusión de librerías.

7 Orden de ejecución.

Explicados todos los bloques, para poder poner en marcha el simulador hay que tener en cuenta la secuencia de ejecución de los programas:

- 1- En la ventana de comandos de Ubuntu se debe ejecutar: “roslaunch unity_simulation_scene unity_simulation_scene.launch”. Dicha instrucción lanzará el nodo unity_simulation_scene, el rosbridge socket y podrán publicar y suscribirse a todos los Topics, como se observa en la Figura 39.



```
/home/strudermey/catkin_ws/src/unity_simulation_scene/launch/unity_simulation_scene.launch http://localhost:11311
file_server (file_server/file_server)
rosapi (rosapi/rosapi_node)
rosbridge_websocket (rosbridge_server/rosbridge_websocket)

auto-starting new master
process[master]: started with pid [1033]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to a5206d88-efef-11ea-bb59-309c238d77b2
process[rosout-1]: started with pid [1044]
started core service [/rosout]
process[rosbridge_websocket-2]: started with pid [1047]
process[rosapi-3]: started with pid [1048]
process[file_server-4]: started with pid [1049]
[ INFO] [1599362326.279088100]: ROSbridge File Server initialized.
2020-09-06 05:19:02+0200 [-] Log opened.
2020-09-06 05:19:06+0200 [-] registered capabilities (classes):
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.call_service.CallService
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.advertise.Advertise
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.publish.Publish
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.subscribe.Subscribe
2020-09-06 05:19:06+0200 [-] - <class 'rosbridge_library.capabilities.defragmentation.Defragment'>
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.advertise_service.AdvertiseService
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.service_response.ServiceResponse
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.unadvertise_service.UnadvertiseService
2020-09-06 05:19:06+0200 [-] WebSocketServerFactory starting on 9090
2020-09-06 05:19:06+0200 [-] Starting factory <autobahn.twisted.websocket.WebSocketServerFactory object at 0x7f7b12b95a10>
2020-09-06 05:19:06+0200 [-] [INFO] [1599362346.769973]: Rosbridge WebSocket server started at ws://192.168.56.1:9090
```

Figura 39, lanzamiento de rosbridge.

- 2- Se debe poner en marcha el simulador en Unity o ejecutar el .exe del simulador como se observa en la Figura 40. Al momento de activar el simulador en la ventana anterior se debería de observar que se ha conectado un cliente y que se ha suscrito al Topic cmd_vel como muestra la Figura 41.

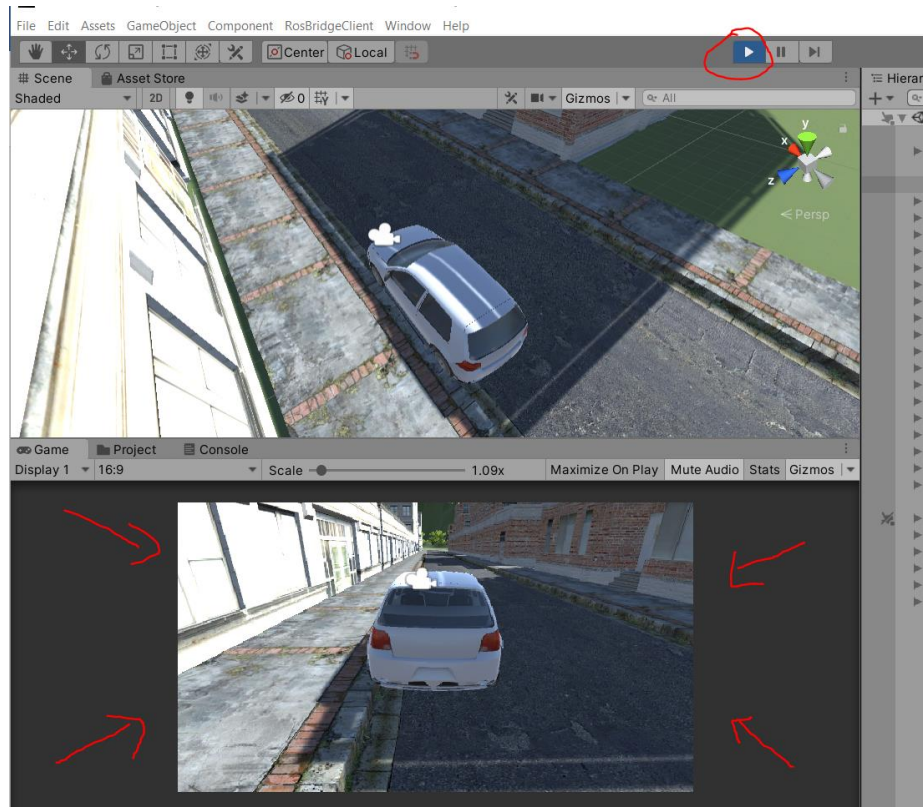


Figura 40, Simulador en marcha.

```

/home/struderm/roscatkin_ws/src/unity_simulation_scene/launch/unity_simulation_scene.launch http://localhost:11311
auto-starting new master
process[master]: started with pid [1033]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to a5206d88-efef-11ea-bb59-309c238d77b2
process[rosout-1]: started with pid [1044]
started core service [/rosout]
process[rosbridge_websocket-2]: started with pid [1047]
process[rosapi-3]: started with pid [1048]
process[file_server-4]: started with pid [1049]
[ INFO] [1599362326.279088100]: ROSbridge File Server initialized.
2020-09-06 05:19:02+0200 [-] Log opened.
2020-09-06 05:19:06+0200 [-] registered capabilities (classes):
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.call_service.CallService
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.advertise.Advertise
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.publish.Publish
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.subscribe.Subscribe
2020-09-06 05:19:06+0200 [-] - <class 'rosbridge_library.capabilities.defragmentation.Defragment'>
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.advertise_service.AdvertiseService
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.service_response.ServiceResponse
2020-09-06 05:19:06+0200 [-] - rosbridge_library.capabilities.unadvertise_service.UnadvertiseService
2020-09-06 05:19:06+0200 [-] WebSocketServerFactory starting on 9090
2020-09-06 05:19:06+0200 [-] Starting factory <autobahn.twisted.websocket.WebSocketServerFactory object at 0x7f7b12b...>
2020-09-06 05:19:06+0200 [-] [INFO] [1599362346.769973]: Rosbridge WebSocket server started at ws://192.168.56.1:9090
2020-09-06 05:23:22+0200 [-] [INFO] [1599362602.653001]: Client connected. 1 clients total.
2020-09-06 05:23:33+0200 [-] [INFO] [1599362613.335908]: [Client 0] Subscribed to /cmd_vel

```

Figura 41, Conexión a rosbridge.

- En la ventana de comandos de Ubuntu ejecutar: `roslaunch tfg_control_script main.py`, lo cual iniciará el nodo `main`.

8 Resultados

El proyecto se ha definido en 3 bloques (entorno gráfico, canal y cerebro), y como se sabe tiene dos modos de funcionamiento, modo simulación y modo entrenamiento, los cuales se han detallado en los capítulos anteriores. Ambos relacionados con el objetivo principal planteado de la creación de un simulador para validar el funcionamiento de una red neuronal.

Antes de evaluar los resultados del proyecto en general, cabe destacar que cada bloque realiza su función individual correctamente:

- El entorno grafico envía correctamente a los scripts de ROS# la información que recolecta por sus entradas de teclado y por la cámara secundaria incorporada envía las imágenes. Si bien el entorno podría mejorarse bien sea ampliando el circuito por donde circula el coche, o incorporando un HUD para que sea más amigable con el usuario o programando un sistema de luces, el desarrollado en este proyecto cumple con las funcionalidades que se especificaron al inicio.
- El canal conformado por ROS# - RosBridge – ROS, establece conexión entre ambos extremos (Unity – Script Control), de manera perfecta. Es una herramienta muy útil, ya que puede ser implementada en diversos proyectos que requieran simular un robot en Unity, puesto que dicho entorno permite representar de manera más realista el ambiente al que se va a enfrentar el robot. En este caso permite probar cómo sería un circuito urbano.
- Por último, el control que realiza el script depende de las decisiones que tome la red neuronal, de modo que según los datos de entrada que se le suministren y cuan bien entrenado esté el modelo la predicción será buena o no.

Una vez evaluado por separado cada bloque integrante del simulador, pasamos a ver cómo se comporta y cuáles fueron los resultados obtenidos. En la Figura 42 aparece el funcionamiento del script de control y de cómo se ejecutarán las secuencias de código dependiendo exclusivamente de un botón que es el de la tecla “t”, que indica si quieres estar en modo entrenar o en modo simulación.

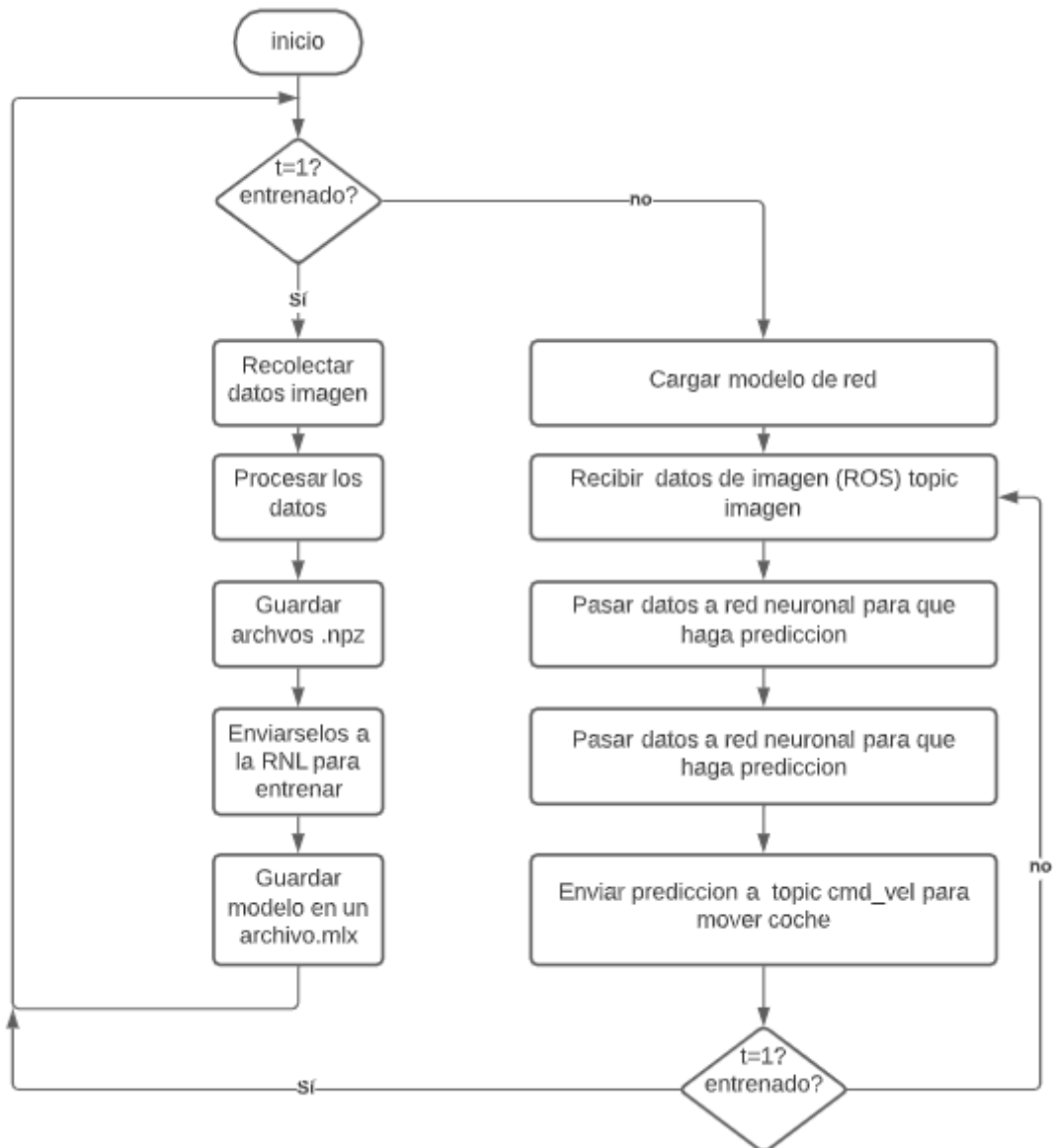


Figura 42, Diagrama de flujo funcionamiento script.

Una vez puesto en marcha el simulador y el script principal se comprobó el modo de funcionamiento de ambos modos, obteniendo como resultado lo que se explica en las siguientes secciones.

8.1 Modo entrenamiento.

Al ejecutar el script y probar el modo entrenamiento, se observa que se ejecutan correctamente todos los pasos, observando cómo se almacenan en la carpeta *collect_data_training*. Para obtener los datos, como se ha indicado anteriormente el usuario del simulador, y asegurándose que se está en modo entrenamiento debe conducir por el circuito urbano intentando no salirse de la carretera y puesto que puede ser información

errada que se le transmita a la red neuronal, por tanto reproducir dicho comportamiento. Luego de recolectar los datos los almacena en ficheros tipo npz. Los cuales se han guardado como se puede ver en la Figura 43 para luego ser llamados al momento de entrenar la RNA .Por otro lado, cuando empieza a entrenar vemos, en la Figura 44, cómo carga los datos de los archivos mencionados antes y muestra el tiempo de inicio y el tiempo final de entrenamiento junto con la precisión del entrenamiento. Siendo en el caso de la imagen uno del 51.03% una precisión regular, ya que ha clasificado como correctas la mitad de las imágenes que se le han transferido en esta sesión. Para eso hay que considerar los factores de obtención de datos, que los mismos son enviados con algo de ruido y que el hardware donde se ha realizado el simulador necesitaba más potencia, debido a que a veces colapsaban los servicios por tener abierto demasiados procesos.

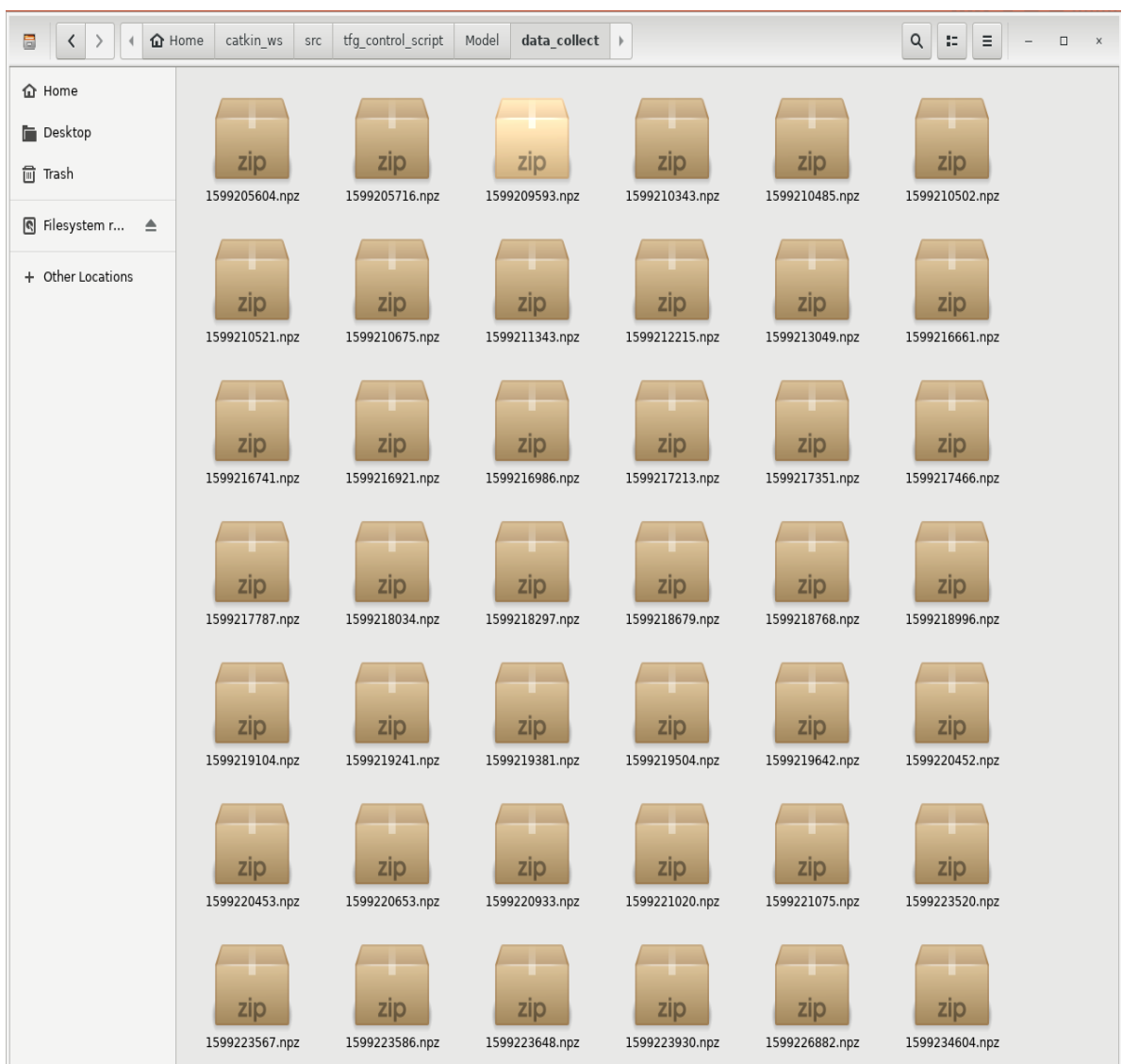


Figura 43, Archivos .npz guardados.

```
struderm@DESKTOP-V4PRJPC: ~
('Label array shape: ', (703, 4))
('Image array shape: ', (786, 153600))
('Label array shape: ', (786, 4))
('Image array shape: ', (786, 153600))
('Label array shape: ', (786, 4))
('Image array shape: ', (786, 153600))
('Label array shape: ', (786, 4))
('Image array shape: ', (786, 153600))
('Label array shape: ', (786, 4))
('Image array shape: ', (786, 153600))
('Label array shape: ', (786, 4))
('Image array shape: ', (786, 153600))
('Label array shape: ', (786, 4))
('Image array shape: ', (786, 153600))
('Label array shape: ', (1056, 4))
('Image array shape: ', (1294, 153600))
('Label array shape: ', (1294, 4))
('Image array shape: ', (1550, 153600))
('Label array shape: ', (1550, 4))
('Image array shape: ', (1550, 153600))
('Label array shape: ', (1550, 4))
('Image array shape: ', (1728, 153600))
('Label array shape: ', (1728, 4))
Loading data duration: 328.29s
Training ...
Training duration: 4634.73s
('Train accuracy: ', '51.03%')
```

Figura 44, Modo entrenamiento en ejecución.

Es normal que tarde tanto tiempo en entrenar puesto que como se menciona anteriormente, el procesamiento de los datos se realiza mientras está todo en marcha (Unity, VM, virtualización de la máquina virtual), y el procesamiento requiere de un hardware de mucha más potencia que la que disponía el equipo donde se realizó el entrenamiento. Pero a efectos bajos se puede comprobar que el script desarrollado funciona puesto que recolecta nuevos datos obtenidos por la cámara del simulador en Unity, luego procesa la imagen, la apila y la guarda en archivos. Posteriormente coge esos datos y entrena a una red sobrescribiendo los datos o bien creando una nueva red desde cero.

8.2 Modo Simulación

Por otro lado, al momento de cambiar al modo simulación y comprobar la red, se realizaron diferentes pruebas:

Primero probando la red preentrenada, que se ha descargado del repositorio. En la Figura 46 se observa que no realiza ninguna predicción, simplemente va recto cuando debería de estar avisando que gire a la derecha para tomar la curva. Esto era lo previsto que no realizara ninguna predicción, debido a que las condiciones en las que se entrenó la red son totalmente diferentes, empezando por el tamaño de la cámara, siguiendo porque el coche estaba en un suelo de color beige y bordeado con hojas blancas para crearle el camino. Cuyo proyecto se puede consultar en el siguiente enlace: <https://github.com/hamuchiwa/AutoRCCar>. Para especificar vemos en la Figura 45, cortesía de “zhaoying9105”. También se puede ver en funcionamiento en el enlace: <https://www.youtube.com/watch?v=BBwEF6WBUQs>.

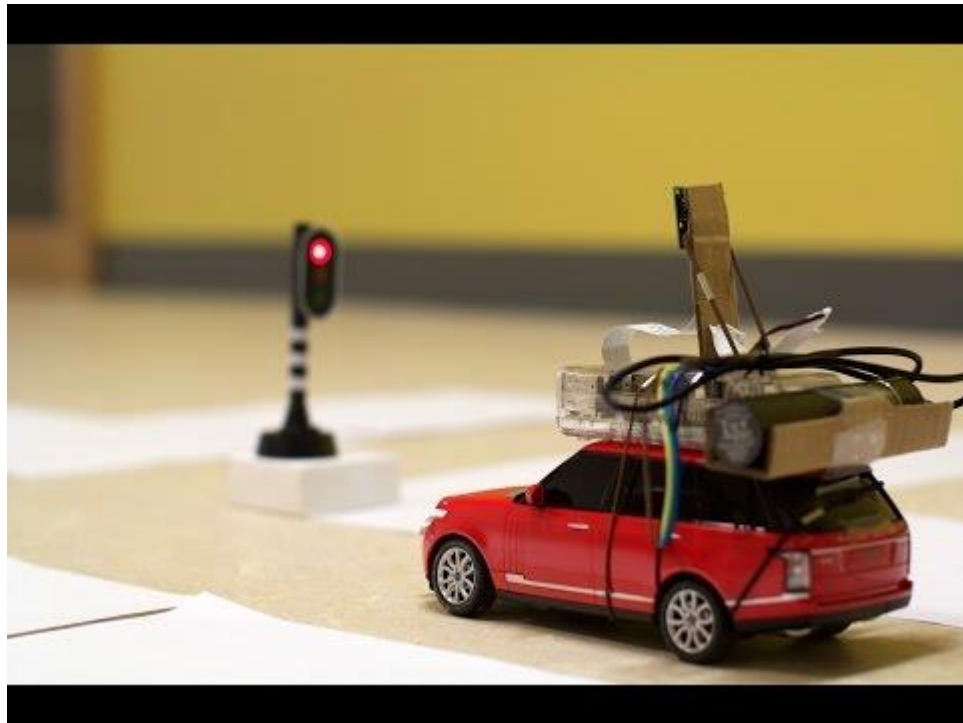


Figura 45, Modelo original de zhaoying9105

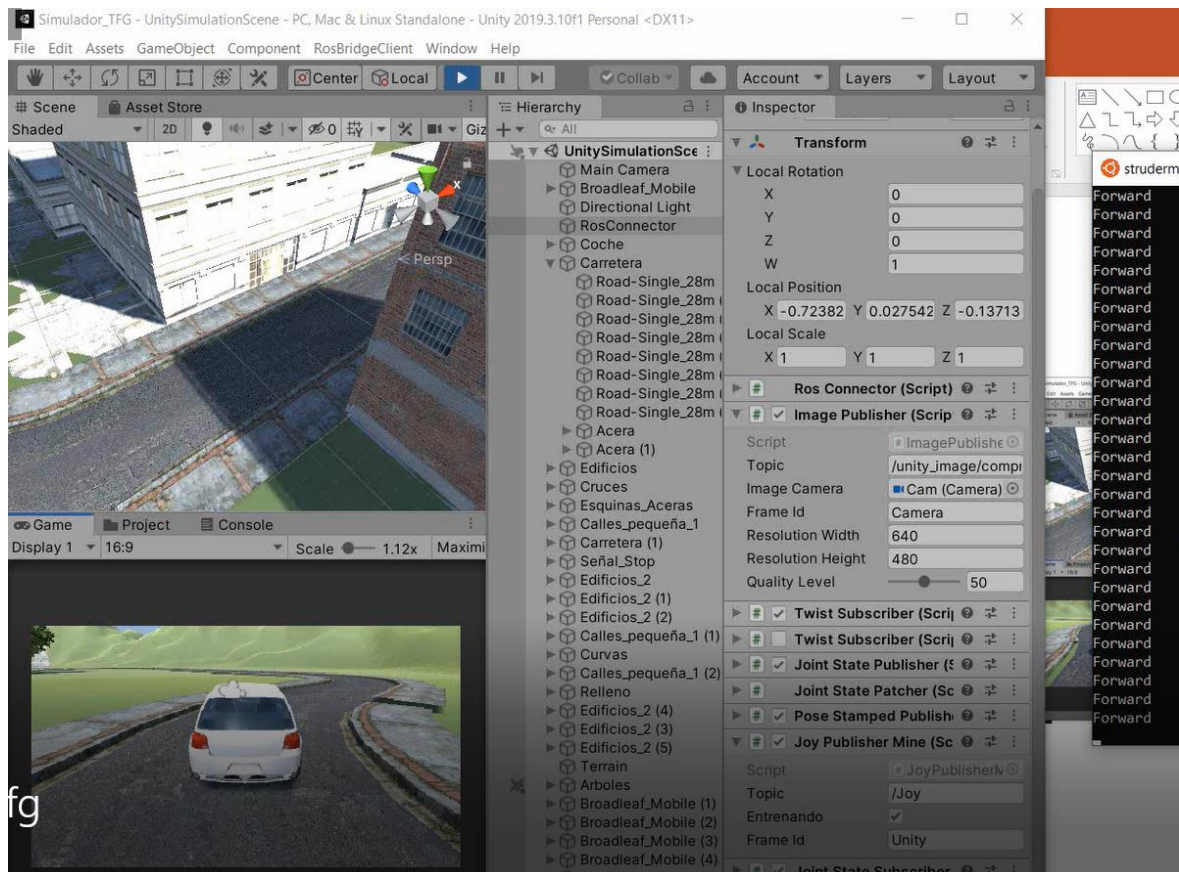


Figura 46, Simulación Red no entrenada

Luego probando la red entrenada desde el simulador. Como vemos en la Figura 47, al inicio le ha costado, siempre daba como resultado ir en línea recta, y salvo en pequeñas ocasiones girar a izquierda o derecha en una curva dependiendo de a qué dirección deba dirigirse. Esto se debe a que a la red le falta entrenamiento o que lo que ha aprendido tiene algunos errores como ruido que se le ha pasado a la imagen por interrupción de la comunicación con rosbriidge que de vez en cuando tiene pérdidas de conexión entre la máquina virtual y el Unity. Si a esto se le añade que al entrenar la red se ha hecho desde un teclado, en el cual la conducción del coche no era óptima, puesto que es complicado conducirlo de manera fluida durante largos periodos, porque el coche al ir a mucha velocidad al girar lleva una inercia dificultando el giro.

Por último, con más tiempo de entrenamiento a la red se ha logrado realizar una mejor aproximación y las curvas las toma de manera más suave. Para ello se ha ajustado que el coche se mueva a una velocidad de 2 m/s que aproximadamente serian 7km/h y no de 50 km/h como estaba programado antes (esto se hace en la Figura 30, Clase *RCControl*. Exactamente en la condición de predicción == 1). Así le da tiempo a la red de tomar la decisión y enviarla mediante ROS.

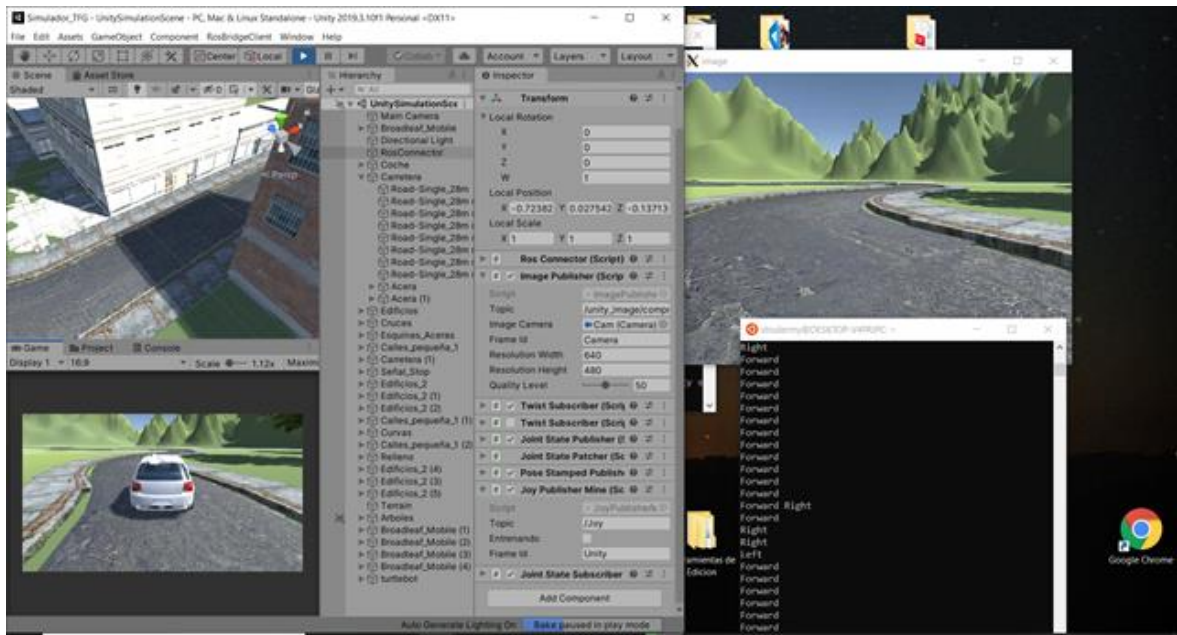


Figura 47, Simulador en modo simulación con red entrenada.

Vistos los resultados, para ambos modos de funcionamiento se puede decir que cumple con el funcionamiento planteado al inicio y que los resultados son positivos abriendo camino a futuras mejoras para una implementación más profunda añadiendo otros elementos que puedan definir a una conducción autónoma. Como por ejemplo un detector de señales de tráfico, simulación de personas transitando y diversos factores que podrían favorecer a la conducción autónoma.

Recopilando toda la información, en el siguiente enlace: <https://youtu.be/mL1X6Zlnw4s> se puede observar de manera práctica el funcionamiento del simulador explicando por pasos ambos modos.

9 Conclusiones

Las posibilidades del desarrollo de algoritmos o la creación de entornos virtuales, que pueden repercutir en el mundo real tienen cada vez más relevancia en el desarrollo tecnológico o es hacia donde se dirige la siguiente generación de la ingeniería, es decir, la implementación de proyectos en el mundo real realizando las pruebas en simuladores para minimizar el error y tener una idea clara de cómo se comportarían sus proyectos. Es por ello que surgió la necesidad de desarrollar un simulador de conducción autónoma con Unity 3D y ROS para el uso de una RNA, La inteligencia artificial comprende uno de los puntos de desarrollo como línea de investigación futura como simulación para la comprensión del comportamiento entre humanos o la interacción hombre-robot y que, por cómo ha sido implementado, podría ampliarse en un futuro de manera independiente al resto del sistema. El proyecto consta de una serie de objetivos y subobjetivos que se han realizado correctamente y permiten obtener una conclusión final.

En primer lugar, se ha abordado exitosamente el objetivo que consta en estudiar las posibles opciones para interconectar un sistema inteligente con un simulador de conducción autónoma con Unity 3D y ROS para uso de red neuronal artificial. A su vez, este objetivo consta de subobjetivos, tal y como recopilar información de distintos sistemas multiagente y simuladores de conducción autónoma la cual se expresa a continuación:

El proyecto se ha definido en 3 bloques (entorno gráfico, canal y cerebro), y como se sabe tiene dos modos de funcionamiento, modo simulación y modo entrenamiento, los cuales se han detallado en los capítulos anteriores. Ambos relacionados con el objetivo principal planteado de la creación de un simulador para validar el funcionamiento de una red neuronal.

Antes de evaluar los resultados del proyecto en general, cabe destacar que cada bloque realiza su función individual correctamente:

El entorno grafico envía correctamente a los scripts de ROS# la información que recolecta por sus entradas de teclado y por la cámara secundaria incorporada envía las imágenes. Si bien el entorno podría mejorarse bien sea ampliando el circuito por donde circula el coche, o incorporando un HUD para que sea más amigable con el usuario o programando un sistema de luces, el desarrollado en este proyecto cumple con las funcionalidades que se especificaron al inicio.

El canal conformado por ROS# - RosBridge – ROS, establece conexión entre ambos extremos (Unity – Script Control), de manera perfecta. Es una herramienta muy útil, ya que puede ser implementada en diversos proyectos que requieran simular un robot en Unity, puesto que dicho entorno permite representar de manera más realista el ambiente al que se va a enfrentar el robot. En este caso permite probar cómo sería un circuito urbano.

Por último, el control que realiza el script depende de las decisiones que tome la red neuronal, de modo que según los datos de entrada que se le suministren y cuan bien entrenado esté el modelo la predicción será buena o no.

Por lo anterior se puede decir que en este proyecto se han expuestos los pilares o bases para futuros estudios e investigaciones basadas en la creación de simuladores que tengan que ver con situaciones reales de la vida real, así mismo, permitir buscar opciones en este caso algoritmos necesarios y otros aspectos necesarios que podrían no necesitar de objetos físicos como robots, como por ejemplo, el reconocimiento de imágenes, un algoritmo de mapeo de la zona en la que se encuentre el robot para lo cual se podría usar sistema como Unity a modo de simulación o el algoritmo de un quadricóptero el cual controle cual es la fuerza o capacidad de giro que debe darse a los rotores para mantenerse estabilizado.

De manera tal, es sin duda un mundo abierto con infinitas posibilidades el de la inteligencia artificial aplicada a mundos 3D como el que se ha creado en este proyecto donde los límites son los establecidos por tu imaginación. A través de los diferentes estados de desarrollo de este proyecto se ha aprendido diferentes tipos de algoritmos de inteligencia artificial, así como de minería de datos, lo importante que es el uso de uno u otro dependiendo de cuál es la motivación del problema que se quiere resolver.

Por otro lado se pudo evidenciar que el uso de algoritmos relacionados con el área de la inteligencia artificial es de vital importancia distinguir qué datos son más importantes que otros o cuales son simplemente inservibles para el propósito de la red neuronal lo cual tiene que ser evaluado por el programador en sí mismo en caso de usar redes neuronales utilizando la implementación del algoritmo de Levenberg-Marquardt el cual no tiene en cuenta outliers por lo que el no tener en consideración detalles como este o simplemente no saber cómo funciona el algoritmo usado en la red neuronal o sus limitaciones llevarán a una solución mediocre, eficiente u optimizada o en el peor de los casos todas a la vez.

.

10 Referencias

- Bischoff, D. (s.f.). *github*. Obtenido de <https://github.com/siemens/ros-sharp/wiki>
docs.ros.org. (15 de 08 de 2020). Obtenido de
http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Joy.html
- DorianKurzaj. (26 de 08 de 2016). *wiki ROS*. Obtenido de <http://wiki.ros.org/Message>
- EAJ, T. (20 de 04 de 2017). *tripackej.blogspot*. Obtenido de
http://tripackej.blogspot.com/2017/04/crear-un-juego-de-coches-en-unity-5_14.html
- Ezquerro, A. (s.f.). *theconstructsim*. Obtenido de <https://www.theconstructsim.com/ros-qa-155-use-callback-function-ros-subscriber/>
- grupo.us.es*. (s.f.). Obtenido de <http://grupo.us.es/gtocom/pid/pid10/RedesNeuronales.htm>
- Mercado, D., Pedraza, L., & Martín, E. (15 de 05 de 2015). *scielo.org.co*. Obtenido de
<http://www.scielo.org.co/pdf/prosp/v13n2/v13n2a11.pdf>
- numpy.es*. (29 de 06 de 2020). Obtenido de
<https://numpy.org/doc/stable/reference/generated/numpy.vstack.html>
- opencv sitio web*. (s.f.). Obtenido de
https://docs.opencv.org/3.4/d0/dce/classcv_1_1ml_1_1ANN__MLP.html
- Perñipan, O. (s.f.). *Universidad Politecnica de Madrid*. Obtenido de
<https://oscarperpinan.github.io/R/ClasesMetodos.pdf>
- zhaoying9105. (s.f.). *github*. Obtenido de <https://github.com/hamuchiwa/AutoRCCar>

11 ANEXOS

11.1 Enlaces GitHub

1. Proyecto original: <https://github.com/hamuchiwa/AutoRCCar>.
2. Paquete Ros#: <https://github.com/siemens/ros-sharp/wiki>.

11.2 Código entero Python

```
#!/usr/bin/env python

import numpy as np

import cv2

import rospkg

import time

import os

import rospy

import glob

import roslib

from sklearn.model_selection import train_test_split

from cv_bridge import CvBridge

from sensor_msgs.msg import CompressedImage

from sensor_msgs.msg import Joy

from geometry_msgs.msg import Twist

aux = True

class CollectDataTraining (object):

    def __init__(self,input_size):

        self.input_size = input_size

        self.sub_image = rospy.Subscriber("/unity_image/compressed",CompressedImage,self.image_callback)

        self.sub_joy = rospy.Subscriber("/Joy",Joy,self.joy_callback)

        self.saved_frame = 0

        self.total_frame = 0

        self.X = np.empty((0, self.input_size))
```

```

self.y = np.empty((0, 4))
    rospy.init_node('main', anonymous=True)
    self.k = np.zeros((4, 4), 'float')
    for i in range(4):
        self.k[i, i] = 1
def image_callback(self,data):
    self.imagen
CvBridge().compressed_imgmsg_to_cv2(data,desired_encoding='passthrough')
def joy_callback(self,data):
    self.axes= data.axes
    self.buttons = data.buttons
def collect (self):
    global aux
    image = self.imagen
    a=self.axes
    b=self.buttons

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    height, width = gray.shape
    roi = gray[int(height/2):height, :]
    temp_array = roi.reshape(1, int(height/2) *width).astype(np.float32)

    cv2.imshow('image', image)
    cv2.waitKey(1)
    #frame += 1
    self.total_frame += 1

# get input from human driver

```

```

# complex orders
if a[0]==1 and a[1]==1:
    print("Forward Right")
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[1]))
    self.saved_frame += 1

elif a[0]==1 and a[1]==-1:
    print("Forward Left")
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[0]))
    self.saved_frame += 1

elif a[0]==-1 and a[1]==1:
    print("Reverse Right")

elif a[0]==-1 and a[1]==-1:
    print("Reverse Left")

# simple orders
elif a[0]==1:
    print("Forward")
    self.saved_frame += 1
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[2]))

```

```

elif a[0]==-1:
    print("Reverse")

elif a[1]==1:
    print("Right")
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[1]))
    self.saved_frame += 1

elif a[1]==-1:
    print("Left")
    self.X = np.vstack((self.X, temp_array))
    self.y = np.vstack((self.y, self.k[0]))
    self.saved_frame += 1

elif b[0]==1: #guardar datos y salir
    aux = False
    file_name = str(int(time.time()))
    directory =
"/home/strudermey/catkin_ws/src/tfg_control_script/Model/data_collect"
    if not os.path.exists(directory):
    os.makedirs(directory)
    try:
    np.savez(directory + '/' + file_name + '.npz', train=self.X,
train_labels=self.y)
    except IOError as e:
        print e

```



```

def load_data(input_size, path):
    print("Loading training data...")
    start = time.time()

    # load training data
    X = np.empty((0, input_size))
    y = np.empty((0, 4))
    training_data = glob.glob(path)

    # if no data, exit
    if not training_data:
        print("Data not found, exit")
        sys.exit()

    for single_npz in training_data:
        with np.load(single_npz) as data:
            train = data['train']
            train_labels = data['train_labels']
            X = np.vstack((X, train))
            y = np.vstack((y, train_labels))

        print("Image array shape: ", X.shape)
        print("Label array shape: ", y.shape)

    end = time.time()
    print("Loading data duration: %.2fs" % (end - start))

    # normalize data
    X = X / 255.

```

```

# train validation split, 7:3
return train_test_split(X, y, test_size=0.3)

def entrenar():
    input_size = 480 * 320
    data_path
"/home/strudermy/catkin_ws/src/tfg_control_script/Model/data_collect/*.npz"

    X_train, X_valid, y_train, y_valid = load_data(input_size, data_path)

    # train a neural network
    layer_sizes = [input_size, 32, 4]
    nn = NeuralNetwork()
    nn.create(layer_sizes)
    nn.train(X_train, y_train)

    # evaluate on train data
    train_accuracy = nn.evaluate(X_train, y_train)
    print("Train accuracy: ", "{0:.2f}%".format(train_accuracy * 100))

    # evaluate on validation data
    validation_accuracy = nn.evaluate(X_valid, y_valid)
    print("Validation accuracy: ", "{0:.2f}%".format(validation_accuracy * 100))

    # save model
    model_path
"/home/strudermy/catkin_ws/src/tfg_control_script/Model/save_model/nn_model.xml"
    nn.save_model(model_path)

```

```

class NeuralNetwork(object):
    def __init__(self):
        self.model = None

    def create(self, layer_sizes):
        # create neural network
        self.model = cv2.ml.ANN_MLP_create()
        self.model.setLayerSizes(np.int32(layer_sizes))
        self.model.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP)
        self.model.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM, 2, 1)
        self.model.setTermCriteria((cv2.TERM_CRITERIA_COUNT, 100, 0.01))

    def train(self, X, y):
        # set start time
        start = time.time()

        print("Training ...")
        self.model.train(np.float32(X), cv2.ml.ROW_SAMPLE, np.float32(y))

        # set end time
        end = time.time()
        print("Training duration: %.2fs" % (end - start))

    def evaluate(self, X, y):
        ret, resp = self.model.predict(X)
        prediction = resp.argmax(-1)
        true_labels = y.argmax(-1)
        accuracy = np.mean(prediction == true_labels)
        return accuracy

```

```

def save_model(self, path):
    directory = "/home/strudermey/catkin_ws/src/tfg_control_script/Model"
    if not os.path.exists(directory):
        os.makedirs(directory)
    self.model.save(path)
    print("Model saved to: " + "" + path + "")

def load_model(self, path):
    if not os.path.exists(path):
        print("Model does not exist, exit")
        sys.exit()
    self.model = cv2.ml.ANN_MLP_load(path)

def predict(self, X):
    resp = None
    try:
        ret, resp = self.model.predict(X)
    except Exception as e:
        print(e)
    return resp.argmax(-1)

class RCDriverNNOnly(object):

    def __init__(self, model_path):

        self.sub_image
        rospy.Subscriber("/unity_image/compressed", CompressedImage, self.image_callback) =
        self.sub_joy = rospy.Subscriber("/Joy", Joy, self.joy_callback)
        rospy.init_node('main', anonymous=True)
        # load trained neural network
        self.nn = NeuralNetwork()

```

```

self.nn.load_model(model_path)

self.rc_car = RCControl()

def joy_callback(self,data):
    self.axes= data.axes
    self.buttons = data.buttons

def image_callback(self,data):
    self.imagen
    CvBridge().compressed_imgmsg_to_cv2(data,desired_encoding='passthrough')

def drive(self):
    global aux
    image = self.imagen
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    b=self.axes

    # lower half of the image
    height, width = gray.shape
    roi = gray[int(height/2):height, :]

    cv2.imshow('image', image)
    # cv2.imshow('mlp_image', roi)

    # reshape image
    image_array = roi.reshape(1, int(height/2) * width).astype(np.float32)

    # neural network makes prediction
    prediction = self.nn.predict(image_array)
    self.rc_car.steer(prediction)

```

```
if b[0]==0:  
    aux = 0
```

```
class RCControl(object):
```

```
    def __init__(self):
```

```
        self.pub_twist = rospy.Publisher('cmd_vel', Twist,queue_size = 10)
```

```
        self.vel_msg = Twist()
```

```
    def steer(self, prediction):
```

```
        global aux
```

```
        if prediction == 2:
```

```
            self.vel_msg.linear.z = 0.0
```

```
            self.vel_msg.linear.x = 2
```

```
            self.vel_msg.linear.y = 0.0
```

```
            self.vel_msg.angular.x = 0
```

```
            self.vel_msg.angular.y = 0
```

```
            self.vel_msg.angular.z = 0
```

```
            print("Forward")
```

```
        elif prediction == 0:
```

```
            self.vel_msg.angular.x = 1
```

```
            self.vel_msg.angular.y = 0
```

```
            self.vel_msg.angular.z = -2
```

```
            print("Left")
```

```
        elif prediction == 1:
```

```
            self.vel_msg.angular.x = 1
```

```
            self.vel_msg.angular.y = 0
```

```
            self.vel_msg.angular.z = 2
```

```

        print("Right")
    else:
        self.vel_msg.linear.x = 0
        self.pub_twist.publish(self.vel_msg)

if __name__ == '__main__':
    s = 480 * 320

    mp=
"/home/struderm/roscatkin_ws/src/tfg_control_script/Model/save_model/nn_model.xml"
    while True:
        ctd=CollectDataTraining(s)
        if aux:
            while aux:
                ctd.collect()
            entrenar()
        else:
            car=RCDriverNNOnly(mp)
            while not aux:
                car.drive()

```