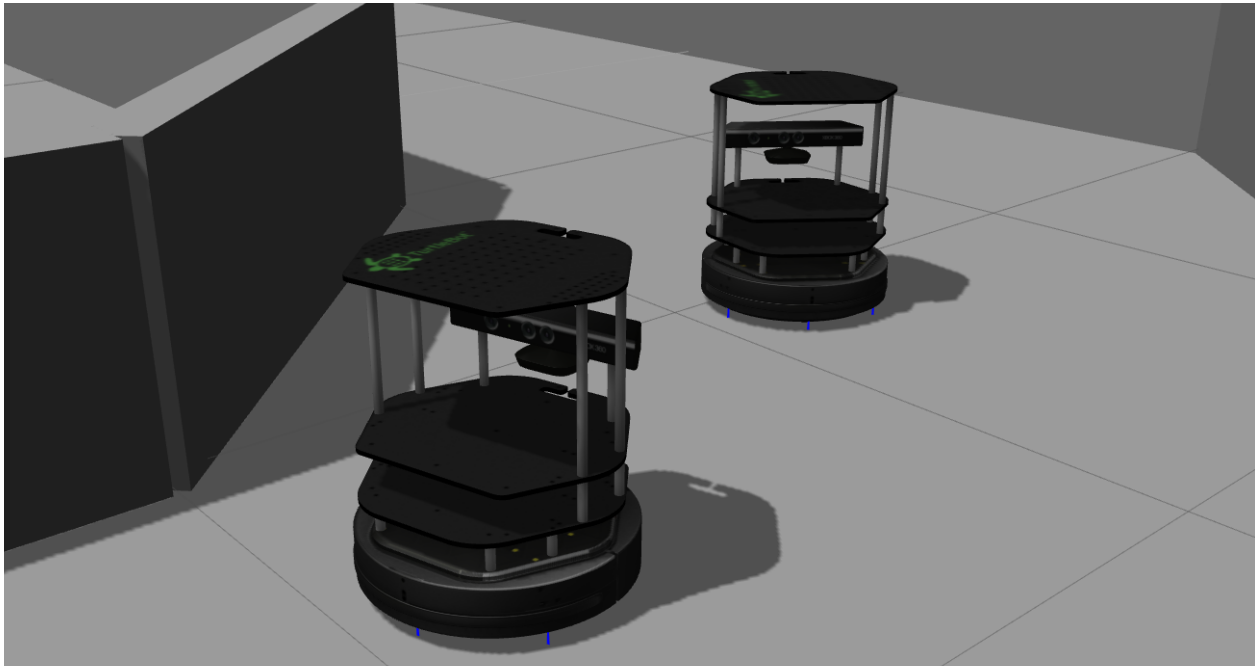


VISIÓN ARTIFICIAL Y ROBÓTICA

PRÁCTICA 1: Carrera de robots



Aarón Picó Pascual - picoaaron@gmail.com

Samuel Arévalo Cañestro - sac52@alu.ua.es

Índice

1. Introducción	3
1.1. Soluciones a la práctica:	3
1.2. Herramientas utilizadas:	3
1.3. Puesta en marcha	4
2. Aproximación	5
2.1. Apartado de navegación	5
Obtener un dataset de imágenes	5
Subir el dataset	8
Construir la red	9
Entrenar la red	10
Guardar el modelo o los pesos	12
Cargar la red en un nodo de ROS (en python)	13
Pasar la imagen de la cámara a la red y los comandos de velocidad al robot	15
2.2. Apartado de detección de otros robots	17
Obtener el dataset de imágenes	18
Construir la red	18
Subir el dataset	18
Cargar la red en el nodo ROS (en python)	18
2.3. Apartado de nube de puntos	20
Calibración de las cámaras	20
Captura de imágenes	23
Generación de nube de puntos	24
3. Experimentación	26
3.1. Cooperación de dos redes	26
Nuevo algoritmo:	27
4. Resultados	29
4.1. Videos del movimiento del robot	29
4.2. Resultados de generación de nubes de puntos visualizados en MeshLab	29
5. Conclusiones	32
6. Referencias	33

1. Introducción

La práctica consta de tres partes:

1. El robot complete de forma totalmente autónoma una vuelta al circuito
2. Calcular el mapa de profundidad aplicando con las cámaras traseras stereo
3. Detección de otros robots

Siendo la primera parte la principal y para la que se contemplan diferentes aproximaciones como solución:

- Solución basada en Deep Learning
- Solución basada en robótica tradicional
- Solución basada en GMapping o Cartographer
- Otras soluciones (previa consulta con el profesor)

Repositorio en Github:

<https://github.com/Aaron99P/VAR>

Descarga el repositorio en formato zip:

https://drive.google.com/file/d/1jVMMdpUY1pSR1gtlNb3PCzRYqj6Mxl_m

1.1. Soluciones a la práctica:

Nosotros hemos optado por la solución basada en **Deep Learning**.

Mediante una red neuronal convolucional a la que le pasamos las imágenes que capta la cámara del robot, se decide el movimiento del mismo.

También hemos optado por esta solución para el apartado de detección de otro robot, entrenando una segunda red similar

En cuanto a mejoras, para el apartado de detección de otros robots, hemos modificado esta segunda red para que además detecte si el robot está por la izquierda o por la derecha, y trabajando en conjunto con la primera red, nuestro robot recorre el circuito y cuando detecta un robot esquivar al mismo.

1.2. Herramientas utilizadas:

Google Colab	Máquina local	Trabajo en conjunto:
<ul style="list-style-type: none">- Python: 3- Tensorflow: 2.4.1- Keras: 2.4.0	<ul style="list-style-type: none">- Ubuntu: 16.04- Python: 2.7- Tensorflow: 1.12.0- Keras: 2.2.4	<ul style="list-style-type: none">- Github- Discord- Google Colab- Carpetas compartidas de Google Drive

1.3. Puesta en marcha

Importante:

En todos los terminales escribir el siguiente comando en el directorio catkin_ws:

- (La primera vez que ejecutamos en catkin_ws) catkin_make
- source devel/setup.bash

Para la puesta en marcha de nuestro proyecto cargaremos el mundo virtual a utilizar escribiendo en un terminal:

- roslaunch ejemplogazebo create_multi_robot.launch

En un segundo terminal ejecutamos el siguiente comando para lanzar el nodo con las redes neuronales que harán que nuestro robot comience a recorrer el mapa:

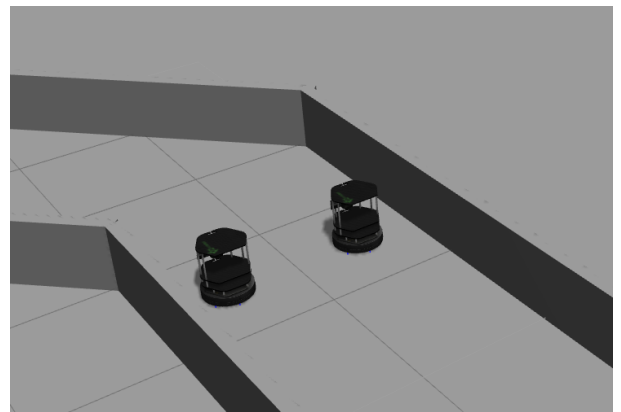
- rosrn python_node red_neuronal.py

Primer terminal:

```
/home/aaron/Escritorio/VAR/catkin_ws/src/ejemplogazebo/launch/create_multi_robot.launch
aaron@aaron-pc:~/Escritorio/VAR/catkin_ws$ roslaunch ejemplogazebo create_multi_robot.launch
... logging to /home/aaron/.ros/log/0fe7b42a-92c3-11eb-85a4-cc2f7167a773/ros-launch-aaron-pc-5734.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

xacro: Traditional processing is deprecated. Switch to --inorder processing!
To check for compatibility of your document, use option --check-order.
For more info, see http://wiki.ros.org/xacro#Processing_Order
xacro.py is deprecated; please use xacro instead
started roslaunch server http://aaron-pc:40077/

SUMMARY
=====
PARAMETERS
* /robot1/depthimage_to_laserscan/output_frame_id: /robot1_tf/camera...
* /robot1/depthimage_to_laserscan/range_min: 0.45
* /robot1/depthimage_to_laserscan/scan_height: 10
* /robot1/robot_pose_ekf/freq: 30.0
* /robot1/robot_pose_ekf/imu_used: False
* /robot1/robot_pose_ekf/odom_used: True
```



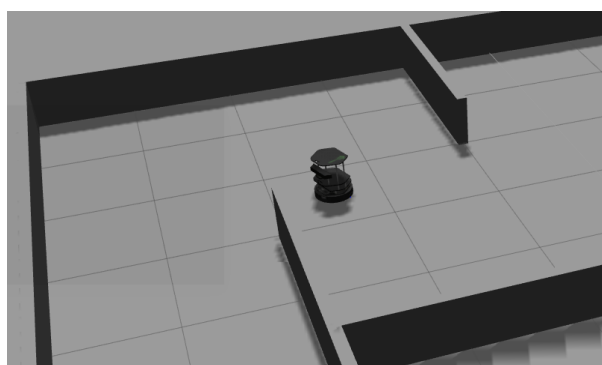
Se iniciará el entorno gazebo y se verá en pantalla el mundo (el circuito) con dos turtle bots en él.

Segundo terminal:

```
aaron@aaron-pc: ~/Escritorio/VAR/catkin_ws
aaron@aaron-pc:~/Escritorio/VAR/catkin_ws$ source devel/setup.bash
aaron@aaron-pc:~/Escritorio/VAR/catkin_ws$ rosrn python_node red_neuronal.py
Using TensorFlow backend.
```

El robot comienza a moverse, podemos comprobar la salida de las redes en este segundo terminal.

```
aaron@aaron-pc: ~/Escritorio/VAR/catkin_ws
pred: Derecha
pred: No veo un Robot
pred: Derecha
pred: No veo un Robot
pred: Derecha
pred: No veo un Robot
pred: Adelante
pred: No veo un Robot
pred: Adelante
pred: No veo un Robot
pred: Adelante
pred: No veo un Robot
pred: Adelante
pred: No veo un Robot
```



2. Aproximación

2.1. Apartado de navegación

Para la solución basada en Deep Learning, necesitamos:

1. Obtener un dataset de imágenes
2. Construir la red
3. Subir el dataset
4. Entrenar la red
5. Guardar el modelo o los pesos
6. Cargar la red dentro de un nodo ROS (en python)
7. Pasar la imagen de la cámara a la red y los comandos de velocidad al robot

Obtener un dataset de imágenes

Estructura del dataset:

- images:
 - |- adelante
 - |- derecha
 - |- izquierda

Capturaremos imágenes de cuando el robot tenga que seguir hacia delante, girar a la derecha y girar a la izquierda; y cada tipo de imagen la guardamos en una subcarpeta diferente.

Al hacerlo de esta manera después podremos cargar el directorio del dataset en Tensorflow.keras y asumirá las imágenes dentro de cada subcarpeta como una misma categoría.

Utilizamos nodos image_view image_saver para guardar las imágenes que captura la cámara del robot.

En cada una de las carpetas ejecutamos:

- `roslaunch image_view image_saver image:=robot1/camera/rgb/image_raw _save_all_image:=false __name:=<nombre>`

Esto generaría en dicha carpeta todas las imágenes que son captadas por la cámara indicada: "robot1/camera/rgb/image_raw", pero no será así por la opción que hemos indicado "_save_all_image:=false", y solo generará las imágenes cuando en otro terminal o mediante código ejecutemos:

- rosservice call /<nombre>/save

Generará una única imagen cada vez que llamemos al servicio.

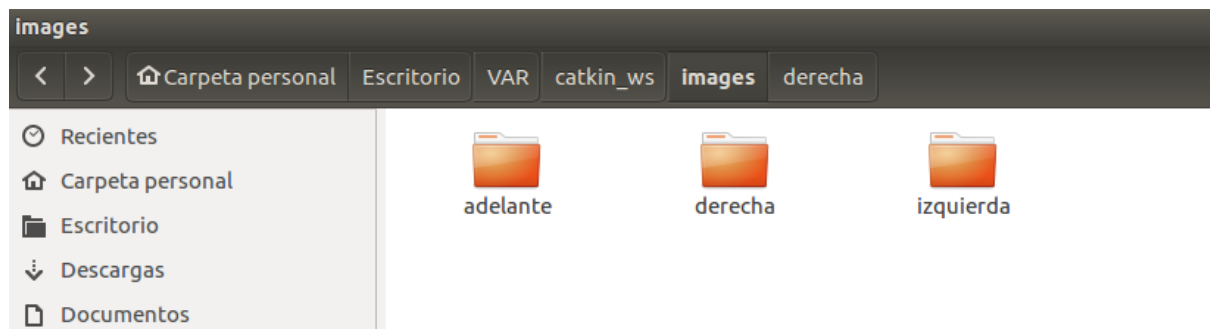
Hemos modificado el fichero main.cpp del nodo send_velocity_commands para ejecutar el comando correspondiente tras mandarle cada orden de movimiento y que así el dataset se genere de manera completamente automática mientras recorres el circuito mandándole por teclado las órdenes al robot:

```

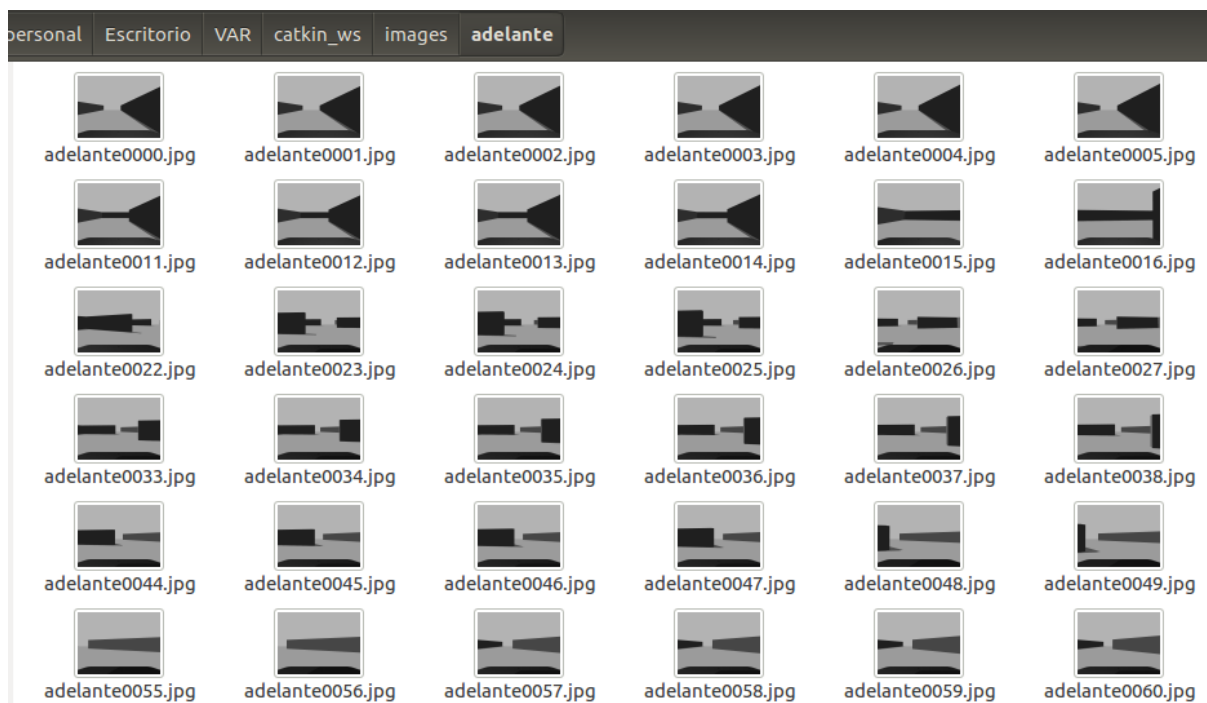
28     bool driveKeyboard()
29     {
30         std::cout << "Type a command and then press enter.  "
31         "Use 'w' to move forward, 'a' to turn left, "
32         "'d' to turn right, '.' to exit.\n";
33
34         //we will be sending commands of type "twist"
35         geometry_msgs::Twist base_cmd;
36
37         char cmd[50];
38         while(nh_.ok()){
39
40             std::cin.getline(cmd, 50);
41             if(cmd[0]!='w' && cmd[0]!='a' && cmd[0]!='d' && cmd[0]!='s' && cmd[0]!='.'){
42                 {
43                     std::cout << "unknown command:" << cmd << "\n";
44                     continue;
45                 }
46
47                 base_cmd.linear.x = base_cmd.linear.y = base_cmd.angular.z = 0;
48                 //move forward
49                 if(cmd[0]=='w'){
50                     base_cmd.linear.x = 0.25;//0.25;
51                     system("rosservice call /image_adelante/save");
52                 }
53                 //turn left (yaw) and drive forward at the same time
54                 else if(cmd[0]=='a'){
55                     base_cmd.angular.z = 0.75;
56                     base_cmd.linear.x = 0.25;
57                     system("rosservice call /image_izquierda/save");
58                 }
59                 //turn right (yaw) and drive forward at the same time
60                 else if(cmd[0]=='d'){
61                     base_cmd.angular.z = -0.75;
62                     base_cmd.linear.x = 0.25;
63                     system("rosservice call /image_derecha/save");
64                 }
65                 //turn right (yaw) and drive forward at the same time
66                 else if(cmd[0]=='s'){
67                     base_cmd.linear.x = -0.25;
68                     system("rosservice call /image_atras/save");
69                 }
70                 //quit
71                 else if(cmd[0]=='.'){
72                     break;
73                 }
74                 //publish the assembled command
75                 cmd_vel pub_.publish(base_cmd);

```

Dataset resultado:



Ejemplo imágenes de adelante:



Subir el dataset

El dataset puede ser subido directamente al servidor utilizado en ese momento de Colab, pero por comodidad vamos a utilizar la opción de realizar esto mediante Google Drive.

Subimos el dataset a nuestro Drive, en un directorio que tengamos localizado.

A continuación montamos nuestro directorio de drive con:

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

Y las cargamos en un datagen con lo siguiente:

```
# Definimos el datagen
datagen = ImageDataGenerator(
    rescale=1./255
)

# Le cargamos el dataset de imágenes
inputs = datagen.flow_from_directory(
    "/content/drive/MyDrive/images_var",
    target_size=(32, 32),
    batch_size=batch_size,
    class_mode="categorical"
)
```

Si lo hacemos bien nos dirá el número total de imágenes cargadas y el número de categorías:

```
Found 1152 images belonging to 3 classes.
```


Construir la red

Para la implementación y entrenamiento de la red hemos utilizado Google Colab.

El modelo de la red es el siguiente:

```
activation = "relu"

def cnn_model():
    #
    # Neural Network Structure
    #

    input_shape = (32, 32, 3)

    inputs = keras.Input(shape=input_shape)

    x = layers.Conv2D(52, (5, 5), activation=activation)(inputs)
    x = layers.MaxPooling2D(pool_size=(4, 4))(x)

    x = layers.Conv2D(124, (5, 5), activation=activation)(x)
    x = layers.MaxPooling2D(pool_size=(2, 2))(x)

    x = layers.Flatten()(x)

    x = layers.Dense(120, activation=activation)(x)
    x = layers.Dense(84, activation=activation)(x)
    x = layers.Dense(32, activation=activation)(x)

    outputs = layers.Dense(nb_classes, activation='softmax')(x)

    model = keras.Model(inputs=inputs, outputs=outputs)

    return model
```

Como puede verse, el input_shape es “32, 32, 3”

32, 32 es por la dimensión de las imágenes (vamos a redimensionarlas a 32x32)
3 es porque el color de cada píxel de la imagen está codificado con 3 valores (RGB).

Entrenar la red

Para entrenar la red instanciamos el modelo, hacemos el compile y lo entrenamos con la función fit.

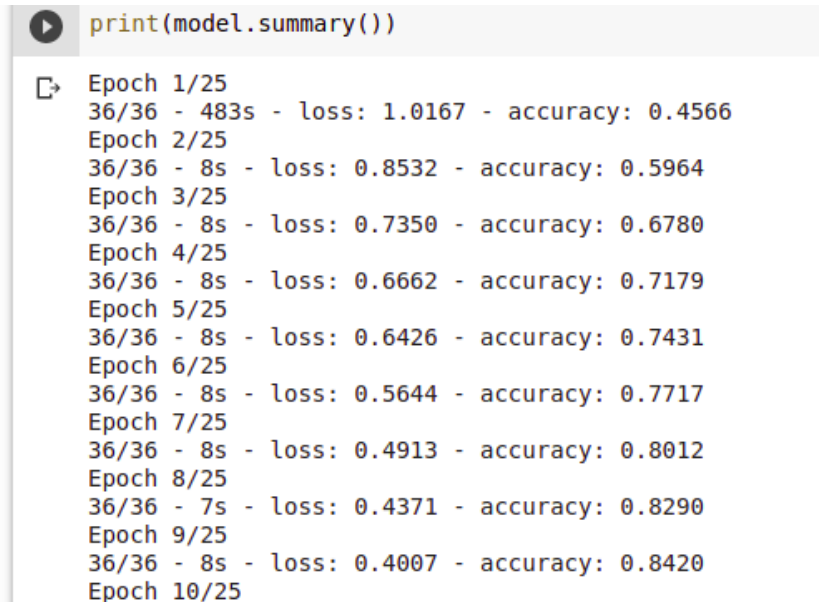
```
model = cnn_model()

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
early_stopping = EarlyStopping(monitor='accuracy', patience=4)

model.fit(inputs, epochs=25, batch_size=32, verbose=2, callbacks=early_stopping)

print(model.summary())
```

Al ejecutarlo vemos como comienza a entrenar y el valor de loss y accuracy que tiene en cada epoch.



```
print(model.summary())
```

Epoch	36/36	Time	loss	accuracy
1/25	36/36	483s	1.0167	0.4566
2/25	36/36	8s	0.8532	0.5964
3/25	36/36	8s	0.7350	0.6780
4/25	36/36	8s	0.6662	0.7179
5/25	36/36	8s	0.6426	0.7431
6/25	36/36	8s	0.5644	0.7717
7/25	36/36	8s	0.4913	0.8012
8/25	36/36	7s	0.4371	0.8290
9/25	36/36	8s	0.4007	0.8420
10/25				

Es cierto que la buena práctica de Deep Learning sería disponer de un conjunto de test para comprobar si el entrenamiento está siendo efectivo viendo si es capaz de clasificar las imágenes de test, las cuales ha visto durante su entrenamiento, o mejor con un cross validation como 10 k-fold CV; pero como el objetivo de la práctica no es este; no nos centraremos en esta parte.

En su lugar, para comprobar que medianamente la red ha entrenado correctamente (aunque no es cómo se debería en un entorno real) cogemos un bloque de imágenes de cada tipo, y comprobamos la salida de la red con ellas:

Siendo las categorías:

- 0: Ir hacia delante
- 1: Girar a la derecha
- 2: Girar a la izquierda

```
def predict(file):
    x = load_img(file, target_size=(32, 32))
    x = img_to_array(x)
    x = np.expand_dims(x, axis=0)
    array = model.predict(x)
    result = array[0]
    answer = np.argmax(result)
    if answer == 0:
        print("pred: Adelante")
    elif answer == 1:
        print("pred: Derecha")
    elif answer == 2:
        print("pred: Izquierda")
    return answer

print("Adelante:")
for i in range(30):
    if(i<10):
        print(predict("/content/drive/MyDrive/images_var/adelante/left000"+str(i)+".jpg"))
    else:
        print(predict("/content/drive/MyDrive/images_var/adelante/left00"+str(i)+".jpg"))

print()
print("Derecha:")
for i in range(17):
    if(i<10):
        print(predict("/content/drive/MyDrive/images_var/derecha/left000"+str(i)+".jpg"))
    else:
        print(predict("/content/drive/MyDrive/images_var/derecha/left00"+str(i)+".jpg"))

print()
print("Izquierda:")
for i in range(30):
    if(i<10):
        print(predict("/content/drive/MyDrive/images_var/izquierda/left000"+str(i)+".jpg"))
    else:
        print(predict("/content/drive/MyDrive/images_var/izquierda/left00"+str(i)+".jpg"))
```

Que nos devuelve algo como lo siguiente, y vemos que realmente es capaz de clasificar las imágenes:

```
Adelante:
pred: Adelante
0
pred: Adelante
0
pred: Adelante
0
[...]
```

```
Derecha:
pred: Derecha
1
pred: Derecha
1
pred: Derecha
1
[...]

Izquierda:
pred: Izquierda
2
pred: Izquierda
2
pred: Izquierda
2
[...]

```

Esto, aunque sea de una manera burda, nos permite saber que nuestra red ha entrenado.

Y probaremos si de verdad la red es eficaz directamente viendo el desempeño del robot con ella.

Guardar el modelo o los pesos

Debemos tener una manera de poder guardar la red ya entrenada para después poder usarla cuando la necesitemos. La red es entrenada una sola vez, luego es usada todas las veces que se requiera.

Podemos guardar el modelo y los pesos de la siguiente manera, siendo model la instancia de la red que hemos entrenado:

```
model.save('./modelo/modelo.h5')
model.save_weights('./modelo/pesos.h5')
```

Cargar la red en un nodo de ROS (en python)

No vamos a poder cargar el modelo directamente por la diferencia de versiones de Tensorflow y Keras utilizadas en Colab y en nuestra máquina local con ROS, debido a que en la misma debemos utilizar las compatibles con python 2.7 (no podemos usar el 3 por incompatibilidad con herramientas de ROS).

Las versiones de cada una están indicadas en la introducción.

Pero sí podemos cargar nuestra red entrenada a pesar de esto, gracias al fichero con los pesos y de la siguiente manera:

No podemos cargar el modelo mediante el archivo, pero nada nos impide crearlo de nuevo siguiendo la misma estructura, por lo que replicaremos el método para crear nuestro modelo que teníamos en Colab:

```
65 def cnn_model(nb_classes):
66     #
67     # Neural Network Structure
68     #
69
70     activation = "relu"
71
72     input_shape = (32, 32, 3)
73
74     inputs = keras.Input(shape=input_shape)
75
76     x = layers.Conv2D(52, (5, 5), activation=activation)(inputs)
77     x = layers.MaxPooling2D(pool_size=(4, 4))(x)
78
79     x = layers.Conv2D(124, (5, 5), activation=activation)(x)
80     x = layers.MaxPooling2D(pool_size=(2, 2))(x)
81
82     x = layers.Flatten()(x)
83
84     x = layers.Dense(120, activation=activation)(x)
85     x = layers.Dense(84, activation=activation)(x)
86     x = layers.Dense(32, activation=activation)(x)
87
88
89     outputs = layers.Dense(nb_classes, activation='softmax')(x)
90
91     model = keras.Model(inputs=inputs, outputs=outputs)
92
93     return model
94
```

Y dentro de la clase red neuronal la instanciamos:

```
97 class red_neuronal:
98
99     global cmd_vel_pub #publisher para mandar las velocidades al robot
100
101     longitud, altura = 32, 32
102
103     #Modelo para la navegacion
104     pesosNavegacion = './pesosNavegacion.h5'
105     modelNavegacion = cnn_model(3)
106     modelNavegacion.load_weights(pesosNavegacion)
107
```

Entonces, como se ve en la imagen superior, a esa instancia creada del modelo le cargamos los pesos del fichero de pesos que hemos obtenido en Colab tras entrenar la red con la función “model.load_weights(ficheroConLosPesos)”

Pasar la imagen de la cámara a la red y los comandos de velocidad al robot

Lo primero, y por eso hacemos en el inicializador de la clase, será declarar los subscribers y publishers necesarios:

```
115
116 def __init__(self):
117     #El CvBridge lo utilizaremos para obtener las imagenes OpenCV
118     self.bridge = CvBridge()
119     #image_sub será el subscriber mediante el que obtendremos los mensajes (imagenes) desde la cámara
120     self.image_sub = rospy.Subscriber("robot1/camera/rgb/image_raw", Image, self.callback)
121     #cmd_vel_pub será el publisher mediante el que enviaremos mensajes (velocidades) a la mobile_base del robot
122     self.cmd_vel_pub = rospy.Publisher("/robot1/mobile_base/commands/velocity", Twist)
123
124
```

La función asociada al subscriber de la cámara es callback. En ella recibimos un mensaje de imagen que convertimos a imagen con OpenCV, pasamos la misma al método para predecir qué movimiento debe realizarse, el resultado lo enviamos al método velocidad(resultadoDeLaRed) que nos devolverá el mensaje con las velocidades que enviaremos acto seguido al mobile_base del robot utilizando el método publish del publisher.

```
195 | #Método asociado al subscriber image_sub de la cámara
196 def callback(self, image_message):
197     try:
198         #Obtenemos la imagen
199         cv_image = self.bridge.imgmsg_to_cv2(image_message, desired_encoding='bgr8')
200         cv2.waitKey(3)
201
202         #La red decide que movimiento hacer
203         pred = self.predictNavegacion(cv_image)
204
205         #Calculamos la velocidad que vamos a enviar al robot
206         base_cmd = self.velocidad(pred)
207
208         #Enviamos la velocidad
209         self.cmd_vel_pub.publish(base_cmd)
210
211         #La red de deteccion dice si hay o no un robot
212         det = self.predictDeteccion(cv_image)
213
214     except CvBridgeError as e:
215         print(e)
216
```

La función predictNavegación redimensiona la imagen que le es pasada como parámetro, así encajará con el input_shape de la red (32,32,3). Tras esto se pasa al método predict del modelo y devolvemos el número de categoría con el valor más alto en el array devuelto. Es decir, la posición del array con un número más alto se corresponde con el número de la categoría que es predicha por la red.

```
124
125 def predictNavegacion(self, f):
126     #Redimensionamos la imagen a la que necesita la red
127     img = cv2.resize(f, (self.longitud, self.altura))
128     x = img_to_array(img)
129     x = np.expand_dims(x, axis=0)
130
131     #Usamos el modelo predict del modelo
132     with graph.as_default():
133         self.modelNavegacion._make_predict_function()
134         array = self.modelNavegacion.predict(x)
135         result = array[0]
136
137     #Del array obtenido, la posicion con un
138     #valor mas alto sera la categoria que la
139     #red predice
140     answer = np.argmax(result)
141     #La categoría 0 es seguir recto
142     if answer == 0:
143         print("pred: Adelante")
144     #La categoría 1 es girar a la derecha
145     elif answer == 1:
146         print("pred: Derecha")
147     #La categoría 2 es girar a la izquierda
148     elif answer == 2:
149         print("pred: Izquierda")
150
151     return answer
```


La función `velocidad` recibe como parámetro la categoría predicha (número del 0 al 2) por la red, y se encarga de elaborar el mensaje con las velocidades que se debe enviar al robot en función de dicha categoría.

```
174
175 def velocidad(self, pred):
176     #Este es el tipo de mensaje que debe recibir el robot
177     cmd = geometry_msgs.msg.Twist()
178     cmd.linear.x = cmd.angular.z = 0
179
180     #Si el robot debe seguir recto hacia adelante
181     if pred == 0:
182         cmd.linear.x = 0.5
183     #Si el robot tiene que girar a la derecha
184     elif pred == 1:
185         cmd.angular.z = -0.75
186         cmd.linear.x = 0.25
187     #Si el robot tiene que girar a la izquierda
188     elif pred == 2:
189         cmd.angular.z = 0.75
190         cmd.linear.x = 0.25
191
192     return cmd
193
```

2.2. Apartado de detección de otros robots

Para el apartado de detección de otros robots hemos seguido la misma dinámica que para el recorrido del circuito, utilizando una red neuronal que hemos entrenado mediante un nuevo dataset.

Los pasos a realizar son los mismos que los seguidos para el apartado de navegación (a excepción del último).

1. Obtener el dataset de imágenes
2. Construir la red
3. Subir el dataset
4. Entrenar la red
5. Guardar el modelo o los pesos
6. Cargar la red dentro de un nodo ROS (en python)
7. Pasar la imagen a la red de detección y mostrar si se ve o no un robot

Obtener el dataset de imágenes

Estructura del dataset de detección:

```
images:
  |-no_robot
  |-robot
```

En la categoría no robot hemos reutilizado todas las imágenes que teníamos del dataset destinado a entrenar el recorrido.

Construir la red

La red utilizada sigue la misma estructura que la anterior pero cambiando el número de salidas a 2.

Subir el dataset

Se realiza de la misma manera que en el apartado de navegación. Subiéndolo a nuestra cuenta de Google Drive y luego montándola en el servidor de Google Colab.

Cargar la red en el nodo ROS (en python)

Hemos decidido implementarlo en el mismo nodo que el de navegación. Así es más sencillo desplegarlo todo y no tenemos que replicar código como la creación del modelo u obtener nuevamente la imagen de la cámara.

Lo instanciamos de la misma manera pero pasando como parámetro el número de categorías, o salidas, de la red

```
96
97 class red_neuronal:
98
99     global cmd_vel_pub #publisher para mandar las velocidades al robot
100
101     longitud, altura = 32, 32
102
103     #Modelo para la navegacion
104     pesosNavegacion = './pesosNavegacion.h5'
105     modelNavegacion = cnn_model(3)
106     modelNavegacion.load_weights(pesosNavegacion)
107
108     #Modelo para la deteccion de otro robot
109     pesosDeteccion = './pesosDeteccion.h5'
110     modelDeteccion = cnn_model(2)
111     modelDeteccion.load_weights(pesosDeteccion)
```

En la función callback, asociada al subscriber de la cámara, añadimos una llamada al método de predicción para la detección “predictDeteccion”, pasándole la misma imagen capturada de la cámara que ya habíamos pasado al método de predicción de la navegación.

```
195 | #Método asociado al subscriber image_sub de la cámara
196 | def callback(self, image_message):
197 |     try:
198 |         #Obtenemos la imagen
199 |         cv_image = self.bridge.imgmsg_to_cv2(image_message, desired_encoding='bgr8')
200 |         cv2.waitKey(3)
201 |
202 |         #La red decide que movimiento hacer
203 |         pred = self.predictNavegacion(cv_image)
204 |
205 |         #Calculamos la velocidad que vamos a enviar al robot
206 |         base_cmd = self.velocidad(pred)
207 |
208 |         #Enviamos la velocidad
209 |         self.cmd_vel_pub.publish(base_cmd)
210 |
211 |         #La red de deteccion dice si hay o no un robot
212 |         det = self.predictDeteccion(cv_image)
213 |
214 |     except CvBridgeError as e:
215 |         print(e)
```

Esta función sigue la misma estructura y tiene el mismo objetivo que predictNavegacion y, recibiendo una imagen, que redimensionará a las medidas necesarias para la red (32x32) nos devolverá la categoría a la que la red predice que pertenece la imagen.

```
153 | def predictDeteccion(self, f):
154 |     #Redimensionamos la imagen a la que necesita la red
155 |     img = cv2.resize(f, (self.longitud, self.altura))
156 |     x = img_to_array(img)
157 |     x = np.expand_dims(x, axis=0)
158 |
159 |     with graph.as_default():
160 |         self.modelDeteccion._make_predict_function()
161 |         array = self.modelDeteccion.predict(x)
162 |         result = array[0]
163 |
164 |     #Del array obtenido, la posicion con un
165 |     #valor mas alto sera la categoria que la
166 |     #red predice
167 |     answer = np.argmax(result)
168 |     if answer == 0:
169 |         print("pred: No veo un Robot")
170 |     elif answer == 1:
171 |         print("pred: VEO UN ROBOT")
172 |
173 |     return answer
174 |
```

2.3. Apartado de nube de puntos

En este apartado se intentará generar una nube de puntos a partir de dos imágenes estéreo capturadas por las cámaras traseras de nuestro robot. Para ello deberemos:

1. Calibrar las cámaras para obtener los valores necesarios
2. Capturar las imágenes
3. Generar la nube de puntos empleando los datos de calibración y las imágenes obtenidas

Calibración de las cámaras

Para calibrar las cámaras traseras del robot introduciremos en nuestro mundo de gazebo un panel de tablero de ajedrez. Las imágenes en las que aparezca este panel nos servirán para aplicar el algoritmo de calibración.

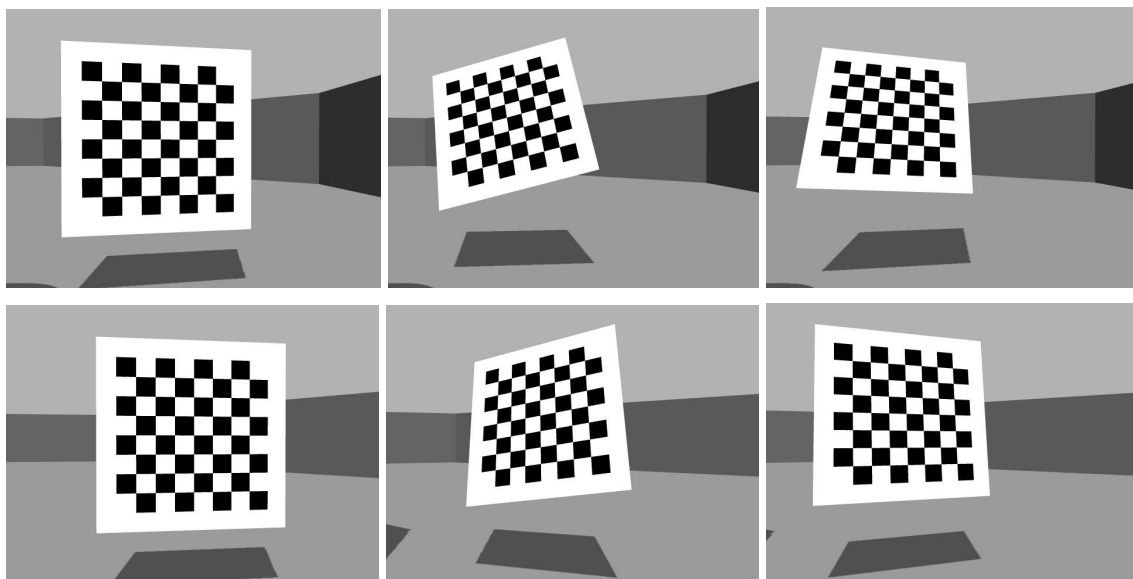
Generaremos un tablero 8 x 8 empleando un algoritmo de creación de tableros para gazebo, este algoritmo se encuentra recogido en:

<https://gist.github.com/Kukanani/4b09debf29eafdd4d96c4717520e6f18>

Capturaremos varias imágenes tal y como se ha comentado anteriormente con cada una de las cámaras traseras. Lo haremos ejecutando el comando:

```
roslaunch image_view image_saver image:=<camara_a_elegir>
```

Tras tener varias imágenes de nuestro tablero desde ambas cámaras, en posiciones relativamente distintas, podemos proceder con la calibración.



Para conseguir los valores de calibración de las cámaras, que posteriormente serán empleados para generar nubes de puntos, emplearemos un algoritmo de calibración basado en el tutorial de calibración de OpenCV[3].

En primer lugar importamos numpy, cv2, glob y sys, ya que nos harán falta durante el algoritmo.

```
1  #!/usr/bin/python
2  import numpy as np
3  import cv2
4  import glob
5  import sys
6
```

Tras esto, se han definido varias variables que sirven para tener un acceso más rápido a las rutas de las imágenes que se emplearán durante la calibración. Estas rutas se obtienen desde línea de comandos al ejecutar el script.

```
8  #::Ubicaciones de las imagenes empleadas en la calibracion
9  camara=str(sys.argv[1])
10 image_resolve = str(sys.argv[2])+'/'+camara+'/'+camara+'_0001.jpg'
11 image_path = str(sys.argv[2])+'/'+camara+'/*.jpg'
12
```

El algoritmo comienza definiendo los criterios de finalización de la calibración, seguido de la preparación de vectores de puntos que serán usados para almacenar las esquinas detectadas en las imágenes de nuestro tablero.

```
13 #::Criterio para finalizacion del algoritmo de calibracion
14 criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
15
16 # Preparacion de vectores de puntos empleados durante la calibración
17 objp = np.zeros((7*7,3), np.float32)
18 objp[:, :2] = np.mgrid[0:7,0:7].T.reshape(-1,2)
19
20 # Arrays empleandos para almacenar puntos de todas las imagenes, de todas las iteraciones
21 objpoints = [] # Puntos en el espacio 3d
22 imgpoints = [] # Puntos en la imagen 2d
23
```

Ahora se obtiene una lista de los ficheros de imagen que se encuentran en la ruta indicada, y comienza el bucle de obtención de puntos. Este bucle intentará detectar esquinas en el tablero y se añadirán dichos puntos a los vectores anteriormente mencionados.

```
25 images = glob.glob(image_path)
26 # Para todas las imagenes en la ruta realizamos el bucle de obtencion de puntos.
27 for fname in images:
28     img = cv2.imread(fname)
29     gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
30
31     # Se buscan las esquinas de los cuadros de calibracion
32     ret, corners = cv2.findChessboardCorners(gray, (7,7),None)
33
34     # Si se encuentran, se añaden a los vectores de puntos 2d y 3d
35     if ret == True:
36         objpoints.append(objp)
37
38         corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
39         imgpoints.append(corners2)
40
41     # Para cada imagen se dibuja la imagen y por encima las esquinas encontradas.
42     img = cv2.drawChessboardCorners(img, (7,7), corners2,ret)
43     cv2.imshow('img',img)
44     cv2.waitKey(500)
45
```

Tras este bucle, se procede a la calibración como tal, empleando la función de opencv “cv2.calibrateCamera”, que estima los valores propios de la cámara que capturó la imagen indicada.

```
46 cv2.destroyAllWindows()
47 #::Calibramos la camara emplenado la funcion de opencv2
48 # Recibe como argumento los arrays de puntos y devvuelve los valores estimados de la camara
49 ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1],None,None)
50
51 #::Probaremos los calores de la camara con una imagen
52 img = cv2.imread(image_resolve)
53 h, w = img.shape[:2]
54 newcameramtx, roi=cv2.getOptimalNewCameraMatrix(mtx,dist,(w,h),1,(w,h))
55
```

Además. probaremos que los valores son correctos, intentando rectificar una imagen empleando los valores y la función cv2.undistort().

```
56 #::Rectificamos la imagen usando la funcion undistort de opencv2 con las valores
57 dst = cv2.undistort(img, mtx, dist, None, newcameramtx)
58
59 # Recortamos los bordes
60 x,y,w,h = roi
61 dst = dst[y:y+h, x:x+w]
62 cv2.imwrite(sys.argv[3]+ '/' +camara+'calibresult.png',dst)
```

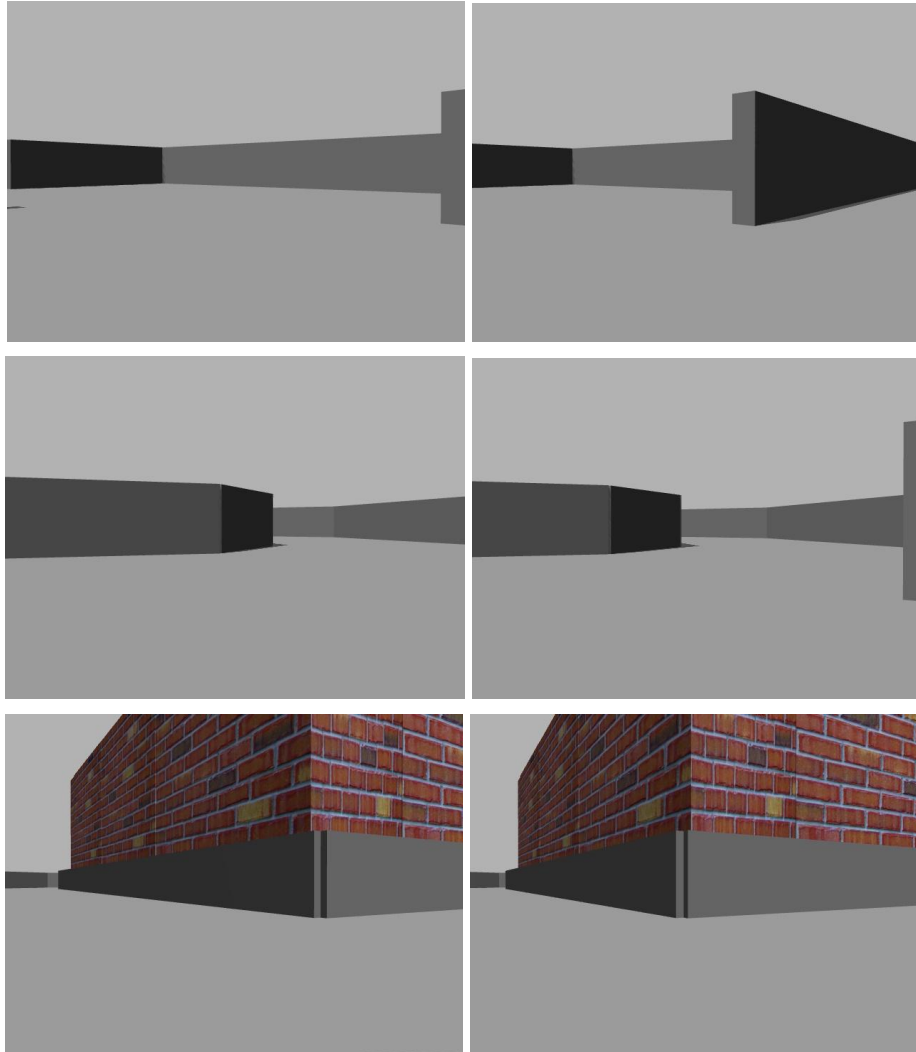
Una vez hecho esto, el script procederá a mostrar y guardar los valores de calibración de la cámara (que son vectores) en ficheros de numpy.

```
62 cv2.imwrite(sys.argv[3]+ '/' +camara+'calibresult.png',dst)
63 print("Dst: " ,dst)
64 print("Ret: " ,ret)
65 print("mtx: " ,mtx)
66 print("dist: " ,dist)
67 print("rvecs: " ,rvecs)
68 print("tvecs: " , tvecs)
69 #::Guardamos los vectores de valores de calibración de la camara en ficheros para u
70 # (fix_imports=True para compatibilidad con Python2)
71 np.save(sys.argv[3]+ '/' +camara+'array_dst.npy',dst,fix_imports=True)
72 np.save(sys.argv[3]+ '/' +camara+'array_mtx.npy',mtx,fix_imports=True)
73 np.save(sys.argv[3]+ '/' +camara+'array_ret.npy',ret,fix_imports=True)
74 np.save(sys.argv[3]+ '/' +camara+'array_dist.npy',dist,fix_imports=True)
75 np.save(sys.argv[3]+ '/' +camara+'array_rvecs.npy',rvecs,fix_imports=True)
76 np.save(sys.argv[3]+ '/' +camara+'array_tvecs.npy',tvecs,fix_imports=True)
77
```

Una vez hemos ejecutado este script tendremos los valores necesarios para proceder con la generación de nubes de puntos.

Captura de imágenes

Mediante Gazebo, moveremos el robot a distintas posiciones y tomaremos capturas desde ambas cámaras. Estas son algunas de las imágenes tomadas:



Generación de nube de puntos

Para generar nubes de puntos a partir de dos imágenes estéreo, se hará uso de dos algoritmos, uno en el que la imagen empleada es la capturada y otro en el que se rectifica la imagen capturada para la generación de la nube de puntos. Ambos algoritmos son exactamente iguales, a excepción de el uso de la función de rectificación en este último.

Nuestro generador de nubes de puntos emplea una función generadora de salidas para transformar los vectores de puntos que forman la nube en un fichero ply que podrá ser visualizado mediante aplicaciones como MeshLab[6].

```
13 #::Funcion para guardar nubes de puntos en ficheros ply
14 def create_output(vertices, colors, filename):
15     colors = colors.reshape(-1,3)
16     vertices = np.hstack([vertices.reshape(-1,3),colors])
17
18     ply_header = '''ply
19         format ascii 1.0
20         element vertex %(vert_num)d
21         property float x
22         property float y
23         property float z
24         property uchar red
25         property uchar green
26         property uchar blue
27         end_header
28     '''
29     with open(filename, 'w') as f:
30         f.write(ply_header %dict(vert_num=len(vertices)))
31         np.savetxt(f,vertices,'%f %f %f %d %d %d')
32
```

La función principal del algoritmo comienza obteniendo las imágenes indicadas y , en la versión con rectificación, emplea la función que rectifica las imágenes.

```
53 #::Funcion principal del algoritmo
54 if __name__ == '__main__':
55     print('Comenzando ...')
56     images = glob.glob(imagedir)
57     # Obtenemos las imagenes desde las que realizamos la nube de puntos
58     imgL = cv2.imread(images[0])
59     imgR = cv2.imread(images[1])
60
61
62     # Usamos la funcion rectifyImage para rectificar las imagenes
63     imgL = rectifyImage(imgL, './camera_calibration/calibration_result/'+camara1+'/')
64     imgR = rectifyImage(imgR, './camera_calibration/calibration_result/'+camara2+'/')
65
```

Esta función de rectificación obtiene los valores de calibración de la cámara (anteriormente obtenidos mediante el script de calibración), y hace uso de las funciones de opencv “getOptimalNewCameraMatrix” y “undistort” para rectificar la imagen.

```
65 #::Funcion dedicada a la rectificacion de una imagen recibida por parametros
66 def rectifyImage(originalImage, calibPath):
67
68     mtx = np.load(calibPath+'array_mtx.npy')
69     dist = np.load(calibPath+'array_dist.npy')
70     h, w = 480, 640
71
72     newcameramtx, roi=cv2.getOptimalNewCameraMatrix(mtx,dist,(w,h),1,(w,h))
73
74     dst = cv2.undistort(originalImage, mtx, dist, None, newcameramtx)
75
76     # Se corta la imagen para retirar los bordes producidos al rectificar
77     x,y,w,h = roi
78     dst = dst[10:470, 10:630]
79
80     # Se devuelve la imagen rectificada
81     return dst
82
```


Tras esto, volvemos a la función principal con las imágenes preparadas para la generación de la nube de puntos.

Se crea un objeto StereoSGBM, con atributos personalizados mediante el cual computamos la matriz de disparidad entre las imágenes.

```
66     # Preparamos el objeto StereoSGBM de opencv2 con atributos personalizados
67     window_size = 3
68     min_disp = 16
69     num_disp = 112-min_disp
70     stereo = cv2.StereoSGBM_create(
71         mode= 3,
72         minDisparity = min_disp,
73         numDisparities = num_disp,
74         blockSize = 3,
75         P1 = 4*3*window_size**2,
76         P2 = 128*3*window_size**2,
77         disp12MaxDiff = 4,
78         uniquenessRatio = 6,
79         speckleWindowSize = 128,
80         speckleRange = 2
81     )
82
83
84     print('Calculando disparidad ...')
85     # Calculamos la disparidad entre las imagenes izquierda y derecha
86     disp = stereo.compute(imgL,imgR).astype(np.float32) /16.0
87
```

Una vez obtenida dicha matriz, se emplea junto con la matriz de la cámara y la distancia focal (obtenidos en la calibración) para generar una nube de puntos dispares.

```
88     print('Generando nube de puntos 3d ...')
89
90     h, w = imgL.shape[:2]
91     # Creamos la nube de puntos a partir de la matriz disp y los valores de calibracio
92     # Estos valores deben haber sido obtenidos en la calibracion de la camara empleand
93     # o mediante otro metodo de obtencion.
94
95     k = np.float32([[554.38271, 0.0, 320.50000],[0.0, 554.38271, 240.50000],[0.0, 0.0,
96     f = k[0]
97     Q = np.float32([
98         [1, 0, 0, -320.50000],
99         [0, -1, 0, 240.50000],
00         [0, 0, 0, -554.38271],
01         [0, 0, 1, 0]])
02
03     # Empleamos las funciones de opencv2 para obtener los vectores de puntos y colores
04     points = cv2.reprojectImageTo3D(disp, Q)
05     colors = cv2.cvtColor(imgL, cv2.COLOR_BGR2RGB)
06
```

Por último, se guardan los puntos de la nube 3D en formato ply usando la función anteriormente mencionada y se informa al usuario del fichero de salida.

```
07     # Guardamos la nube de puntos en un fichero de salida ply para su posterior visualizacion
08     mask = disp > disp.min()
09     out_points = points[mask]
10     out_colors = colors[mask]
11     out_fn = './pointclouds_out/pointcloud_metodo2_out'+images[0].split('/')[2]+' .ply'
12     create_output(out_points, out_colors,out_fn)
13     print('Resultado guardado en fichero de salida: \''+out_fn+'\''')
14
```

3. Experimentación

El acercamiento original ha sido crear una primera red neuronal capaz de dar la vuelta al recorrido, que hace bien, pero no en todos los casos. El dataset se hizo enfocado a seguir el “camino perfecto”, ya que las imágenes fueron extraídas de cómo nosotros lo recorreríamos. Pero esto deja fuera muchos casos en los que el robot puede llegar a encontrarse, tanto porque se desvíe del mismo o porque a conciencia queramos colocarlo en un punto diferente, y en algunos casos ocurre que el robot no es capaz de seguir su recorrido como nos gustaría, llegando incluso a chocarse.

La detección del robot la realizamos con una segunda red neuronal, que lo único que realiza es la clasificación de la imagen para decir si hay o no un robot; pero no se realiza nada con esta información, siendo solo mostrada por terminal.

Por lo tanto hay varias ideas que queremos llevar a cabo ahora para aumentar su desempeño:

1. Aumentar el dataset de imágenes añadiendo nuevos casos, con el fin de que el robot sea capaz de reponerse en casos inesperados
2. Hacer que el robot, además de detectar al segundo robot, sea capaz de esquivarlo

Las propuestas para lograr el segundo punto son dos:

1. Añadir esto a la primera red. Es decir, que la red de navegación aprenda a esquivar el robot
2. Cooperación de dos redes. Pero la de detección del robot tendría tres salidas:
 - a. No hay robot
 - b. Hay un robot a la izquierda
 - c. Hay un robot a la derecha

Vamos a centrarnos en la segunda propuesta, que nos parece más viable por lo expuesto en el siguiente punto.

3.1. Cooperación de dos redes

Ya que ha sido complejo que la red neuronal fuese capaz de hacer el recorrido sin problemas en ningún punto, incluso antes de ponerle la dificultad de también tener en cuenta al segundo robot. La idea más práctica parece ser la segunda, en que las dos redes trabajan en conjunto.

1. En nuestro algoritmo para la navegación primero usaremos la red de detección del robot, y entonces:
 - a. Si esta dice que hay un robot a la derecha, giramos a la izquierda.
 - b. Si esta dice que hay un robot a la izquierda, giramos a la derecha.
 - c. Si esta dice que no hay un robot pasamos al siguiente punto
2. Ahora que sabemos que no hay un robot, utilizamos la red de navegación para saber qué acción tomar como hacíamos en el acercamiento inicial

También debemos tener en cuenta, que siempre que se vea el robot a un lado, no significa que haya que girar al otro; solo lo suficiente para no chocar. Si no, nos desviaremos demasiado. Haremos el nuevo dataset para esta red atendiendo a ello.

Por lo que vamos a conservar la red original que sí será capaz de detectar al segundo robot esté donde esté, solo para poder seguir mostrando esta información. La nueva red estará enfocada solo en esquivar, detectando los casos en los que sí se debe girar debido a encontrarse con un robot en su camino.

Nuevo algoritmo

Como se ha mencionado, ahora el código, tras recibir la imagen (que redimensionará) la pasará primero a la red encargada de saber si hay que esquivar un robot. Si la salida de esta es que no hay que esquivar ningún robot (0), entonces la imagen es pasada a la red encargada de la navegación por el circuito. En cada caso se llamará a un método distinto para generar el mensaje con la velocidad adecuada que enviar al robot.

```
245 #Metodo asociado al subscriber image_sub de la camara
246 def callback(self, image_message):
247     try:
248         #Obtenemos la imagen
249         cv_image = self.bridge.imgmsg_to_cv2(image_message, desired_encoding='bgr8')
250         #Redimensionamos la imagen a la que necesita la red
251         img = cv2.resize(cv_image, (self.longitud, self.altura))
252
253         base_cmd = geometry_msgs.msg.Twist()
254
255         #Red para esquivar
256         esquivar = self.predictEsquivar(img)
257
258         #Si la red de esquivar no ha visto un robot
259         #utilizamos la red de navegacion
260         if esquivar == 0:
261             #La red de navegacion decide que movimiento hacer
262             navegacion = self.predictNavegacion(img)
263             #Calculamos la velocidad que vamos a enviar al robot
264             base_cmd = self.velocidadNavegacion(navegacion)
265         else:
266             #Calculamos la velocidad para esquivar
267             base_cmd = self.velocidadEsquivar(esquivar)
268
269         #Enviamos la velocidad
270         self.cmd_vel_pub.publish(base_cmd)
271
272         #La red de deteccion dice si hay o no un robot
273         detectar = self.predictDeteccion(img)
274
275         cv2.waitKey(3)
276
277     except CvBridgeError as e:
278         print(e)
279
```

Ahora nuestro robot es capaz de esquivar otros robots que se interpongan en su camino, en la mayoría de las situaciones. En escenas complejas donde debe de girar para poder seguir el circuito, si también debe esquivar un robot, suponiendo el giro contrario para esto, a veces no es capaz de complementar ambas y acaba chocando o con la pared o con el robot. Pero en general funciona correctamente.

4. Resultados

4.1. Videos del movimiento del robot

Recorriendo el circuito:

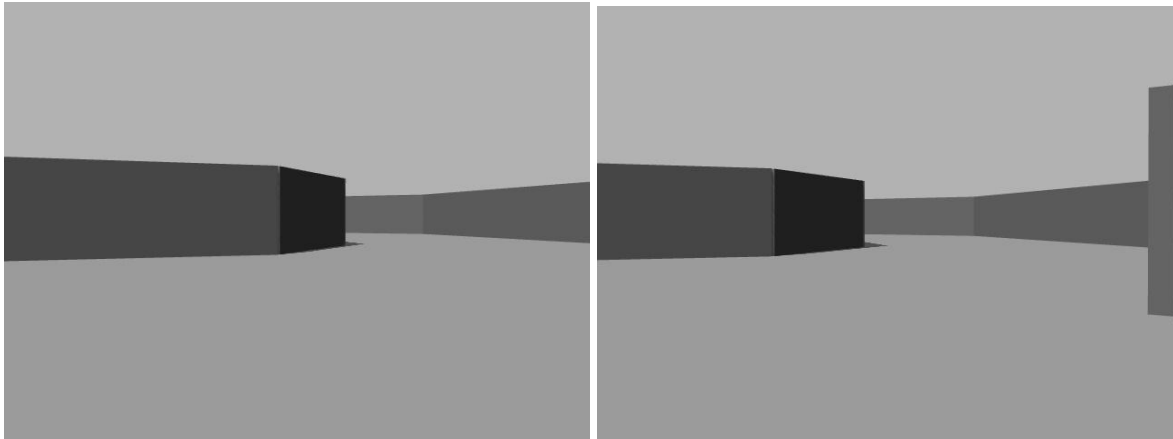
https://www.youtube.com/watch?v=vcaqX7_vOMg

Esquivando otros robots:

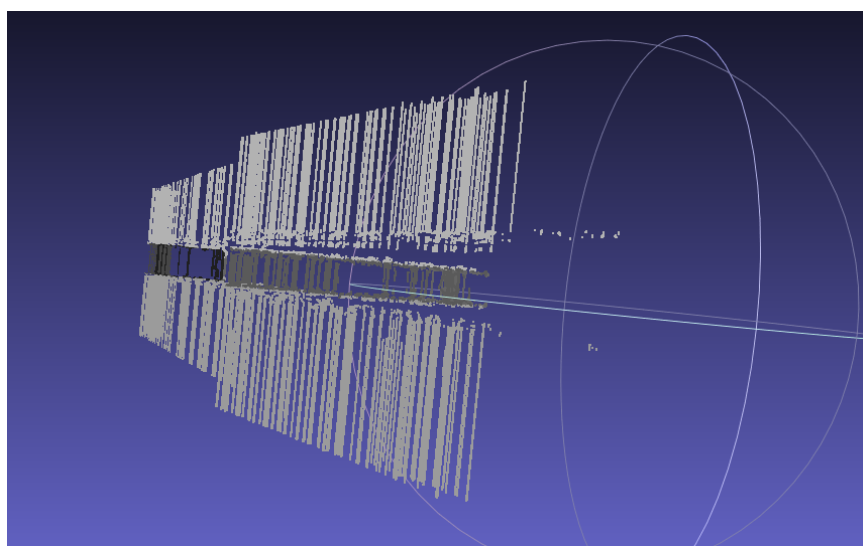
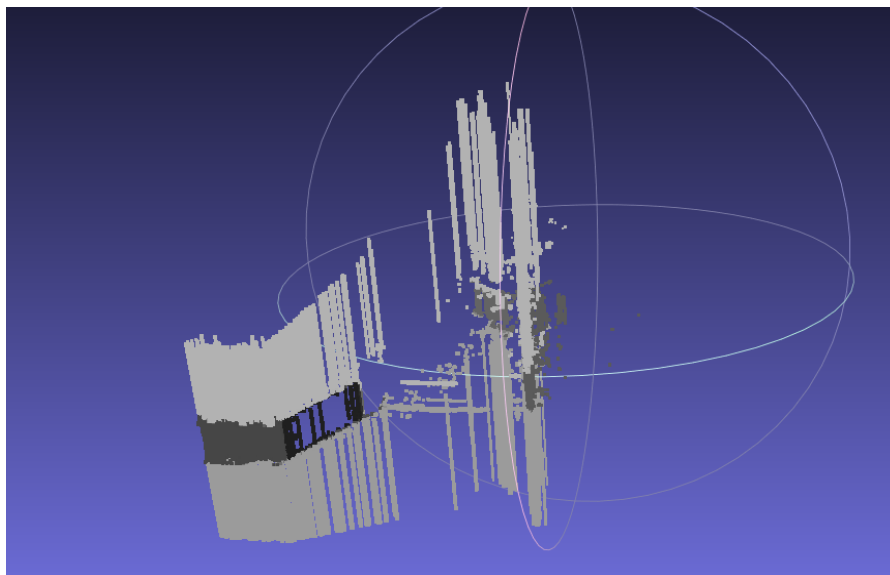
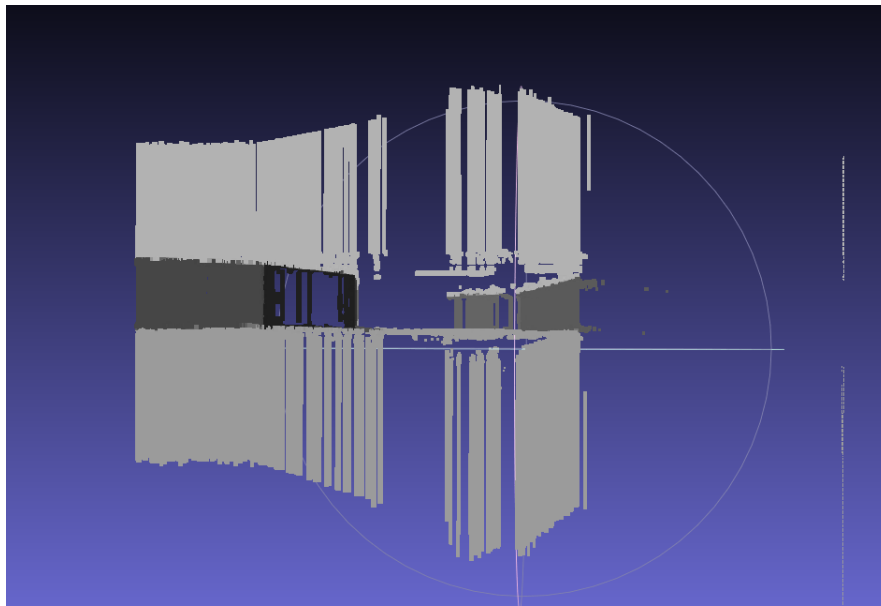
<https://www.youtube.com/watch?v=shUOUVy02vw>

4.2. Resultados de generación de nubes de puntos visualizados en MeshLab

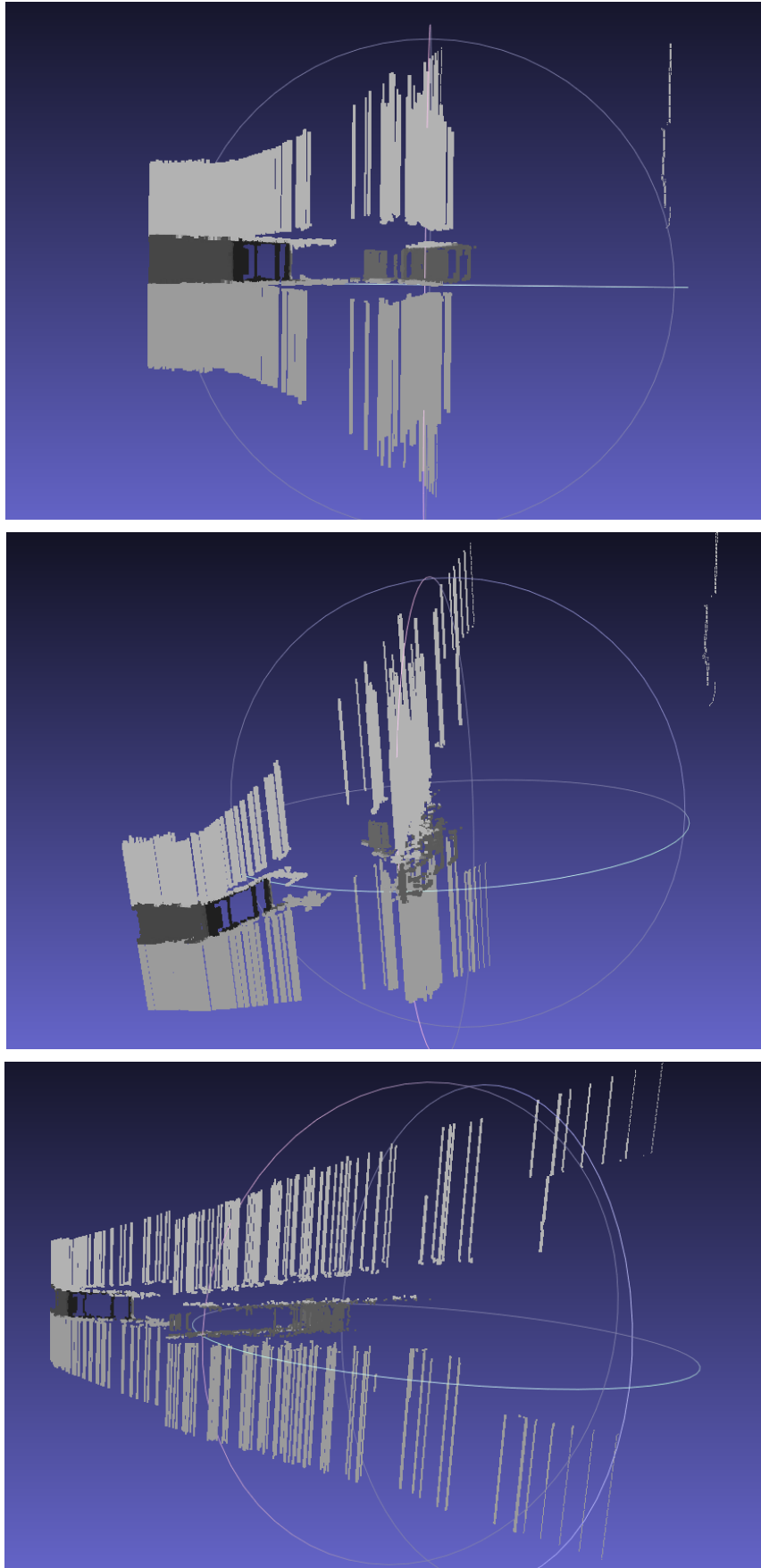
Imágenes originales:



Algoritmo sin rectificación de imagen:



Algoritmo con rectificado de imagen:



5. Conclusiones

A pesar de haber tenido algunas complicaciones durante la realización de la práctica, se ha conseguido sacar muy buenos resultados en cuanto al movimiento del robot en el circuito.

Empleando nuestro conocimiento de redes neuronales adquirido en este último año hemos sido capaces de desarrollar un sistema de movimiento autónomo capaz de dar vueltas al circuito.

En cuanto al procesamiento de imágenes a posteriori y generación de nubes de puntos, los resultados podrían ser mejores, ya que algunas imágenes no generaban nubes de puntos coherentes, cosa extraña ya que había otras que sí se han obtenido sin errores.

Como conclusión, en esta práctica hemos puesto a prueba nuestros conocimientos obtenidos sobre deep learning y aprendido a emplear métodos de manipulación de imágenes con opencv. Además de contar con un entorno de simulación cercano al mundo real, que nos ha permitido tener una experiencia muy similar a lo que sería trabajar con un robot real.

6. Referencias

- [1] <http://wiki.ros.org/> - 02/04/2021
- [2] <http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29> - 02/04/2021
- [3] https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html - 02/04/2021
- [4] https://vovkos.github.io/doxyrest-showcase/opencv/sphinx_rtd_theme/class_cv_StereoSGBM.html - 02/04/2021
- [5] http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython - 02/04/2021
- [6] <https://www.meshlab.net/> - 02/04/2021