Visión Artificial y Robótica
Práctica 1

Alejandro Galán Cuenca
Jorge Donis del Álamo

4 de abril de 2021

Índice

1. Introducción

El propósito de esta práctica es completar el siguiente circuito

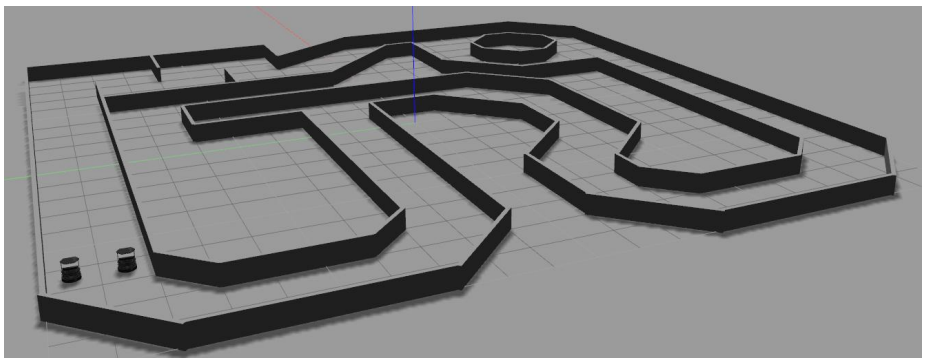


Figura 1: Circuito

Usando el robot *turtlebot*, incorporado con sensores LIDAR, y una cámara kinect. En definitiva, se trata de un problema de **navegación**.

Hemos creado un `LEEME.txt` con los comandos necesarios para la correcta ejecución del robot. Está preparado para ejecutar la aproximación que mejores resultados arroja. Completa varias vueltas al circuito.

2. Aproximación

2.1. Navegación

Nuestra primera aproximación fue usar soluciones preexistentes. ROS viene con un stack de navegación que supuestamente soluciona este problema. La estrategia consistía en reconstruir el mapa usando `GMapping`[1] y después navegar usando el stack de navegación. Sin embargo, nos encontramos con algunos problemas a la hora de reconstruir el mapa

Después intentamos atacar el problema desde una perspectiva más tradicional. De esta manera, intentamos hacer un robot más "reactivo". Un robot que reaccionara al entorno, en este caso, a las paredes. El problema de esta aproximación fue **el escaso ángulo de visión de la cámara Kinect**. Con tan sólo 57^{o}, se hacía imposible detectar las paredes inmediatamente contiguas al robot. De alguna manera, se tendría que guardar una reconstrucción de las paredes que habían sido observadas previamente. Es decir, es imposible girar cuando se deja de ver una pared, ya que se deja de ver demasiado pronto. Dada la complicación de esta implementación (demasiados casos y difícil programación), decidimos abandonarla.

[1] `slam_gmapping` es un paquete de ROS que ofrece una implementación de `OpenSlam`. SLAM es una técnica de reconstrucción de mapas basada en ocupación de celdas 2D.

Finalmente optamos por una **red neuronal**. De esta manera, el problema de navegación se convierte en uno de **clasificación**. Queremos predecir, para una entrada X una clasificación Y de entre las tres posibles:

1. Ninguna acción
2. Girar hacia la izquierda
3. Girar hacia la derecha

El robot siempre mantiene una velocidad constante de 0.9 m/s. Simplemente se ha de escoger cuándo girar. La toma de datos se realizó mediante grabaciones de pulsaciones de teclas con `SDL`.

La velocidad de giro es de 0.3 m/s. Se vió que había problemas implementando una función de acelerado o de frenado. La red necesitaba tener memoria para saber si ya había acelerado previamente. Sin esta memoria, aceleraría ad infinitum. Por eso decidimos por quedarnos únicamente con las tres categorías anteriores.

Para resolver el problema de clasificación, empleamos una red neuronal y aprendizaje supervisado.

Las imagenes son preprocesadas de la siguiente forma:

1. Crop superior de 30 píxeles. La porción superior de la imagen suele contener únicamente cielo (no relevante).
2. Reducción de 3 canales de color a 1. Se mantiene toda la información aún en escala de grises.
3. Reescalado a 128 x 64 px.

Probamos numerosas distintas arquitecturas de red. En un principio comenzamos usando captura de la cámara del tópico

`/robot1/camera/rgb/image_raw`

Usando estos datos de entrada (X) obtuvimos una red aceptable que no se chocaba con las paredes, pero tenía dificultades para completar el circuito. El problema de la imagen de este tópico es que **no transmite ningún tipo de información sobre la profundidad**. Por ello, pasamos a usar el siguiente tópico:

`/robot1/camera/depth/image_raw`

En este caso, la carga de imágenes se volvía un poco más complicada en C++:

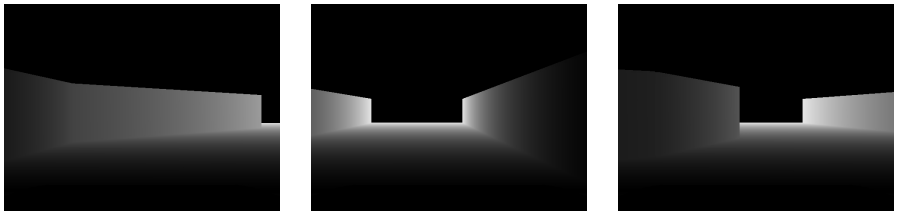
```cpp
void ImageCapturer::image_callback(sensor_msgs::ImageConstPtr const& msg)
{
    auto cv_ptr = cv_bridge::toCvCopy(msg);

    cv::Mat depth_float_img = cv_ptr->image;
    cv::Mat depth_mono8_img;

    if (depth_mono8_img.rows != depth_float_img.rows || depth_mono8_img.cols != depth_float_img.cols)
        depth_mono8_img = cv::Mat(depth_float_img.size(), CV_8UC1);

    cv::convertScaleAbs(depth_float_img, depth_mono8_img, 55, -45);
    this->img = depth_mono8_img;
}
```

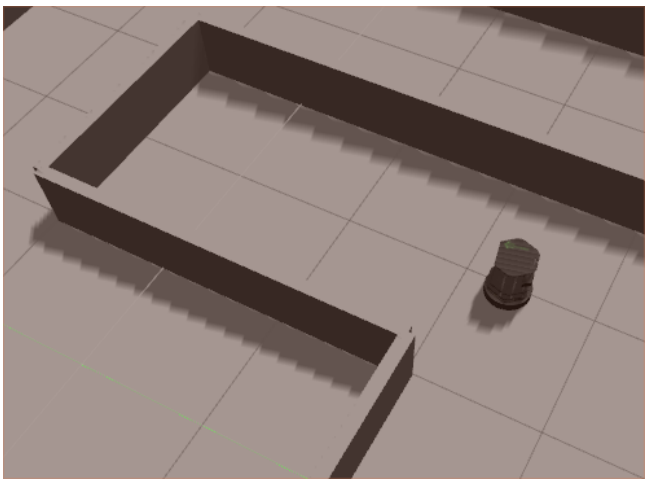
`convertScaleAbs()` transforma la imagen de formato de 16 bytes a 8 bytes (necesario para la red neuronal). Al final se obtenían las siguientes capturas:



NO_ACTION STEER_LEFT STEER_RIGHT

Figura 2: Imágenes con sus categorías

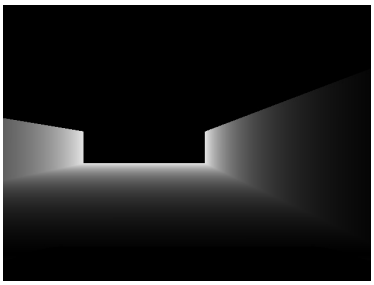
Con esto, la navegación mejoró sustancialmente, pero aún no conseguía completar el circuito. Por ello, nos dimos cuenta de que **necesitábamos más información**. El robot sabía manejarse entre las paredes de los pasillos, pero ciertas acciones concretas del circuito se volvían complicadas. Por ejemplo:



Ese “callejón sin salida” daba problemas. El robot veía las dos paredes e intentaba seguir por el centro. Al final, se acababa chocando. Por ello, vimos necesario algún tipo de dato que nos indicara por qué parte del circuito íbamos. En este caso, usamos **la odometría**. En concreto, capturamos la posición (X, Y) y la inclinación θ. Para obtener la rotación sobre el eje Z, se tiene que hacer una conversión desde el sistema de rotación de **cuaterniones a ángulos de Euler**.

```
tf::Quaternion quat;
tf::quaternionMsgToTF(msg->pose.pose.orientation, quat);
double roll, pitch, yaw;
tf::Matrix3x3(quat).getRPY(roll, pitch, yaw);
```

Con esto, las imágenes se guardaban de la siguiente manera:



$$\underbrace{\texttt{STEER\_LEFT}}_{\text{categoría}}\texttt{\_}\underbrace{\texttt{6.37223}}_{\text{X}}\texttt{\_}\underbrace{\texttt{0.369596}}_{\text{Y}}\texttt{\_}\underbrace{\texttt{-0.0620963}}_{\theta}\texttt{\_}\underbrace{\texttt{1617451835237}}_{\text{timestamp}}\texttt{.jpg}$$

La red, ahora tomaba dos entradas: la imágen y la odometría. La salida seguía siendo una, la acción a realizar.

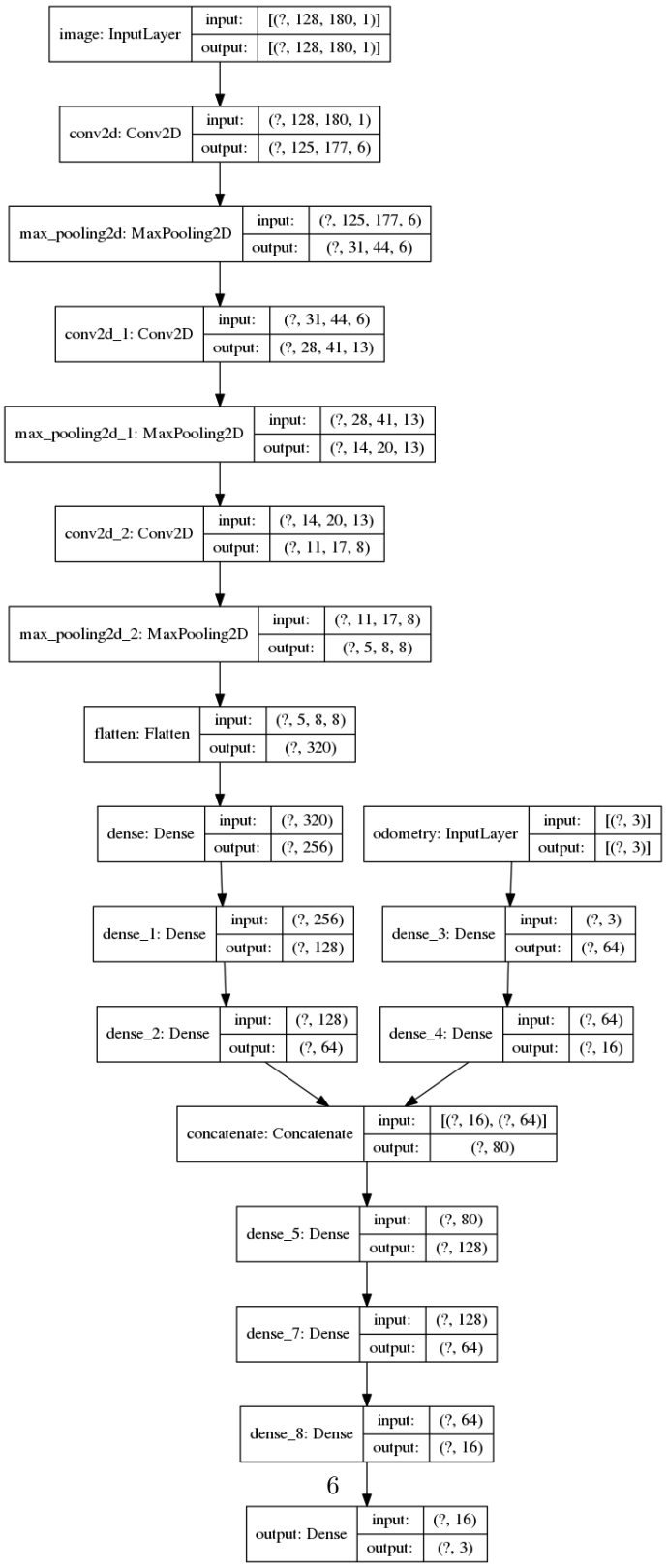


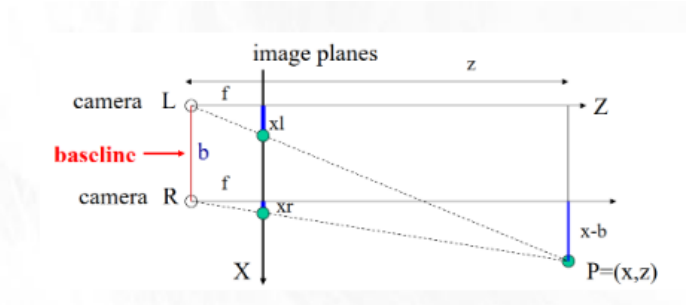
Figura 3: Arquitectura final de la CNN

Una vez encontrada una buena arquitectura, procedimos a grabar datos. Llegamos a grabar más de 30 mil imágenes, aunque fue necesario **equilibrar el número de ejemplos** de las tres categorías. Al final nos quedamos con **14787** imágenes[2]. Conseguimos una precisión sobre el conjunto de validación de **0.79**.

Finalmente, quedaba hacer la predicción en tiempo real para mover al robot. Para ahorrarnos problemas de compatibilidad, optamos por compilar el modelo de Keras directamente en C++14. Esto lo hicimos gracias a la librería **frugally-deep**[3]. Antes de pasar las imágenes a la red neuronal en C++, era necesario hacer el mismo preprocesado que se había hecho en Python durante el aprendizaje.

2.2. Imagen Stereo

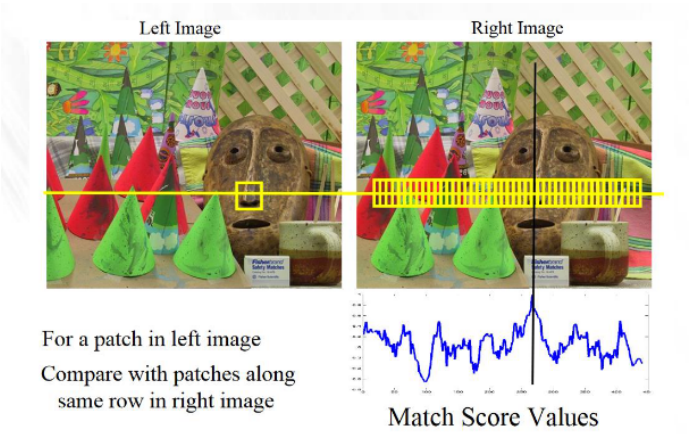
Se intenta conseguir una reconstrucción 3D de la escena a partir de pares de imágenes.



Necesitamos conococer la distancia entre las dos cámaras b y la distancia focal f de cada una de ellas. Con esto, podemos triangular la profundidad z a partir de la proyección del punto real en las dos imágenes (x_l, x_r). Claro está, el problema es que **emparejar** los píxeles de las dos imágenes. Es decir, hay que emparejar los píxeles que corresponden a la misma posición en el mundo real. A esta relación de píxeles se la conoce como **mapa de disparidad**.

Nuestra primera aproximación fue usar el paquete `stereo_img_proc`. Este paquete utiliza el algoritmo *Stereo Block Matching*.

[2] El equivalente a unas 30 vueltas al circuito.
[3] https://github.com/Dobiasd/frugally-deep/



Suponiendo que las dos cámaras se encuentran en el mismo plano epipolar[4], sólo hay que encontrar correspondencia entre píxeles **de la misma horizontal.** Para ello, se busca el mejor emparejamiento dentro de una vecindad. En este caso, la vecindad en un cuadrado.

`stereo_img_proc` obtiene la información del *baseline* (distancia b entre las dos cámaras) y la distancia focal a través del tópico:

`/robot1/trasera/trasera/rgb/camera_info`

Dicho tópico transmite los siguentes paquetes `sensorMsgs/CameraInfo` (obtenidos con `rostopic echo`:

```
header:
  seq: 22152
  stamp:
    secs: 2532
    nsecs: 310000000
  frame_id: "robot1_tf/openni_camera_link"
height: 480
width: 640
distortion_model: "plumb_bob"
D: [0.0, 0.0, 0.0, 0.0, 0.0]
K: [554.3827128226441, 0.0, 320.5, 0.0, 554.3827128226441, 240.5, 0.0, 0.0, 1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [554.3827128226441, 0.0, 320.5, -38.80678989758509, 0.0, 554.3827128226441, 240.5, 0.0, 0.0, 0.0, 1.0, 0.0]
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
  do_rectify: False
```

De aquí, se obtiene la matriz R:

[4]En caso de que las cámaras no se encuentren en el mismo plano, es necesario **rectificar** las dos imágenes.

$$R = \begin{bmatrix} f_x & 0 & c_x & T_x \\ 0 & f_y & c_y & T_y \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

El vector (T_x, T_y) nos indica la **traslación** de la segunda cámara sobre la primera. En definitiva, esconde el valor baseline b.

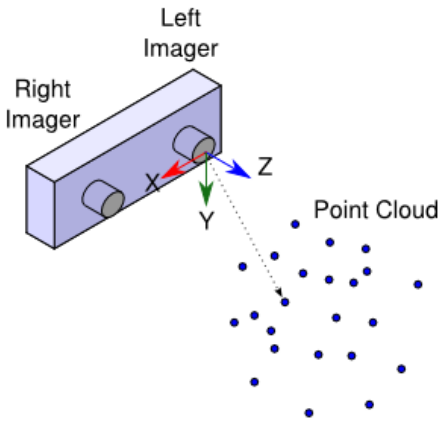
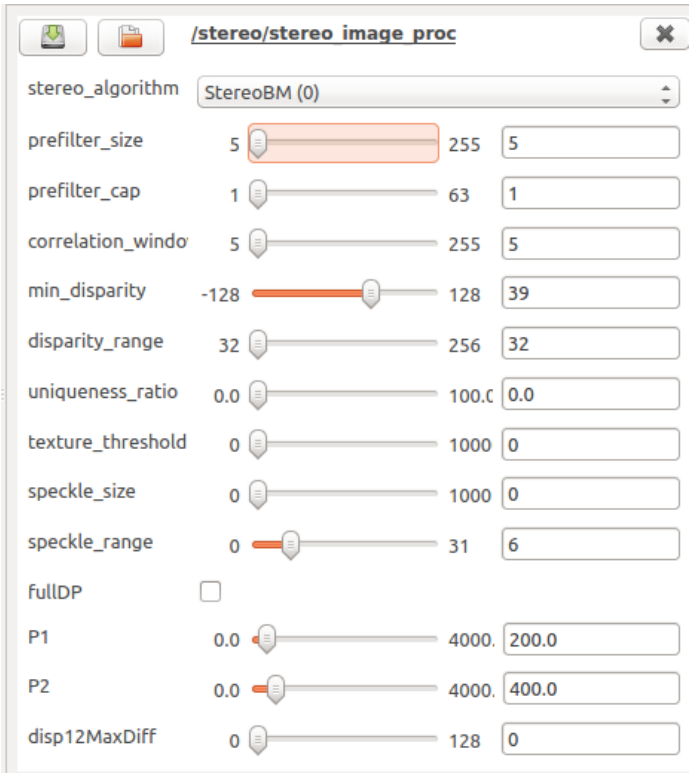


Figura 4: El sistema de referencias se toma sobre la primera cámara (izquierda).

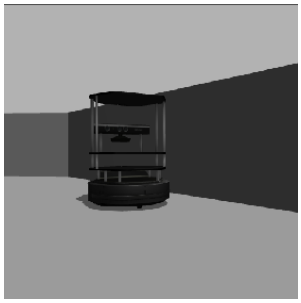
Una vez instalado el paquete, lo lanzamos con el siguiente comando:

```
ROS_NAMESPACE=stereo rosrun stereo_image_proc stereo_image_proc \
left/image_raw:=/robot1/trasera1/trasera1/rgb/image_raw \
left/camera_info:=/robot1/trasera1/trasera1/rgb/camera_info \
right/image_raw:=/robot1/trasera2/trasera2/rgb/image_raw \
right/camera_info:=/robot1/trasera2/trasera2/rgb/camera_info
```

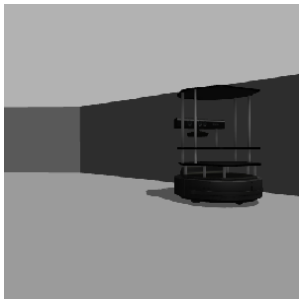
Antes, tenemos que reconfigurar algunos parámetros del algoritmo para obtener buenos resultados. Lanzamos `rqt_reconfigure`:



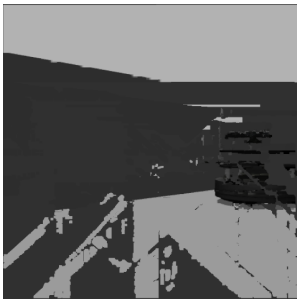
Con ello, obtenemos para cada para de imágenes, una nube de puntos:



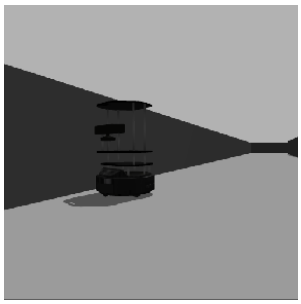
Izquierda



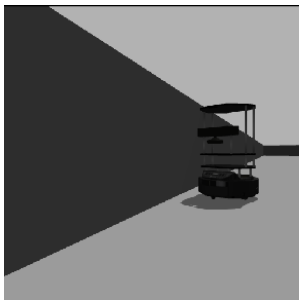
Derecha



Nube de puntos



Izquierda



Derecha

Nube de puntos

Se puede apreciar bastante ruido, y en general, poca profundidad. Esto se debe a que las imagenes no son lo más idóneas. Tal y como veíamos en teoría, **es imposible emparejar píxeles en imágenes sin textura.** Lo mismo se da para imágenes con patrones repetidos. A la hora de buscar posibles emparajemientos en la horizontal, todos los píxeles son válidos, y se pierde la información sobre (x_l, x_r), en definitiva sobre z (profundidad). Esto se podría haber resuelto texturizando las paredes y el suelo, para que dejen de tener un color gris plano.

2.3. Detección de otro Robot

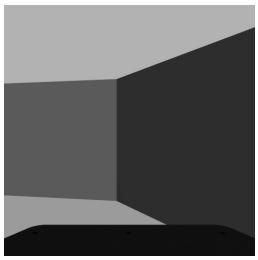
Para realizar esta diferenciación, hemos optado por usar una red neuronal convolucional. La red está inspirida en la arquitectura de la red LeNet5. Se trata de una red con dos capas de filtros de convolución y pooling y dos capas de neuronas activadas por ReLU.

A la hora de preprocesar las imágenes con el objetivo de mejorar la eficiencia de la red (reducir ruido), hemos utilizado opencv. Hemos reducido las dimensiones de 640x480 a 32x32 píxeles. También hemos hecho un crop superior e inferior. Por último, hemos reducido el número de canales de 3 a 1 (escala de grises).

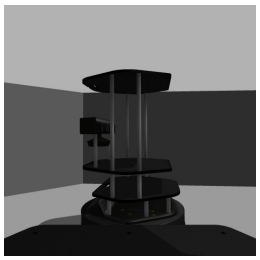
El dataset es barajado, equilibrado y finalmente normalizado (división de todos los píxeles entre 255). Las imágenes de entrenamiento serán un conjunto inicial de imágenes del siguiente tópico:

`/robot1/camera/rgb/image_raw`

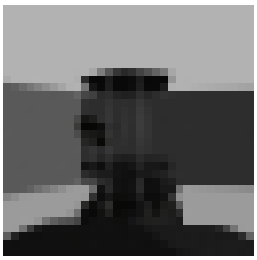
Al final queremos distinguir entre dos categorías: robot y no robot (pared).



Clase de pared



Clase de robot



Reducción de imagen

3. Experimentación

Comenzamos desarrollando varias propuestas, como el uso de GMapping o robótica tradicional, pero no arrojaban resultados. Y, finalmente se tuvo que probar con una solución basada en Deep Learning, la cual nos ha solucionado mejor el problema de dar una vuelta al circuito.

Una vez quedaron fijos los datos de entrada (imágenes de profundidad y odometría), aún quedaba por ver qué arquitectura de red era la más apta. Gracias a la paralelización por GPU, conseguimos hacer muchas pruebas de distintas arquitecturas. Al final mantuvimos aquella con menor error de clasificación en el conjunto de validación.

Otro aspecto a comentar son las capturas de imágenes. Primero hicimos la prueba con la cámara rgb, pero no devolvía resultados convincentes. Por lo que acabamos registrando imágenes de profundidad, y sin cambiar nada más los resultados acaban siendo mucho más positivos.

Finalmente, añadimos simúltáneamente datos de odometría a la red, que le aportan a cada imágen la ubicación x, y, y el ángulo sobre el eje z en el que se encuentra el turtle bot.

Se nos ocurren muchas posibles mejoras a la red neuronal de la navegación. Si bien es cierto que la odometría mejoró la conducción, sigue sin ser suficiente. Las imágenes de profundidad nos dan información sobre el entorno, pero **tienen mucho ruido**. Es decir, la información está **poco discretizada**. Sin embargo, los sensores LIDAR hubieran sido una mejor opción. Es posible que una combinación de sensores LIDAR, odometría e imágenes de profundidad hubiera dado mejores resultados. Por otro lado, respecto a la odometría, cabe mencionar que **no es del todo precisa**. Existe una deriva muy apreciable respecto a la rotación y la posición. Probablemente hubiera sido una buena idea haber prescindido de las posicoines X e Y y haberse quedado únicamente con la rotación en el eje Z. Tampoco cabe descartar la opción de usar la nube de puntos 3D que proporciona la cámara kinect (aunque tampoco hubiera ayudado mucho).

De alguna manera, lo único que se nos ocurre es que el problema no estaba en el modelo de predicción, **sino en los datos**. Esto es porque el error en el conjunto de entrenamiento bajaba sin problema, pero **era muy difícil que la red generalizase**. Incluso reduciendo el número de parámetros libres de la red, se hacía complicado que generalizara. Con 1000 imágenes se obtenía una precisión en el conjunto de validación de 0.87; sin embargo, con 12000 la precisión bajaba a 0.78. Simplemente, los datos no son separables. Es decir, no se puede establecer una relación (o no lo suficientemente precisa) entre la entrada X y la predicción Y. De hecho, si a un ser humano se le presentara el conjunto de imágenes, tampoco sabría clasificarlas correctamente, porque **no existe una relación unívoca**.

Dada una imagen de la pared, no se puede saber a ciencia cierta qué acción tomar (si girar a la izquierda o a la derecha o no hacer nada). Por esto, cuantos más datos grabábamos, peor iba la red. Es posible que esto se hubiera podido solucionar mediante algún tipo de memoria. Por ejemplo, se podrían haber usado series de imágenes, dotando a la red de algún tipo de memoria (capas LTSM).

También se podría haber empleado alguna métrica de error que podría haber sido minimizada mediante aprendizaje por refuerzo. Por ejemplo, se podría haber entrenado (sin supervisión) una red cuyo objetivo es maximizar la distancia entre las dos paredes.

3.1. Red neuronal de detección del otro robot

De la misma forma que la red principal, se eliminarán los duplicados de las imágenes de entrenamiento consiguiento una precisión del **100**. Esto es porque no se utiliza un número elevado de imágenes, cerca de **900**.

No hemos aumentado este número porque resulta suficiente, dado que el ángulo del observador va a ser siempre el mismo, y la diferencia es si capta un objeto en medio o no, sin variar la orientación y el robot siendo la mayor parte del tiempo de un color totalmente diferenciado a la pared, entre otros aspectos. Además optimiza esta situación el preprocesado de imágenes. Por lo tanto arroja buenos resultados sin requerir de tantos datos.

La arquitectura de la red escogida es:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 60, 124, 6)        156
_________________________________________________________________
activation (Activation)      (None, 60, 124, 6)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 30, 62, 6)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 26, 58, 16)        2416
_________________________________________________________________
activation_1 (Activation)    (None, 26, 58, 16)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 29, 16)        0
_________________________________________________________________
flatten (Flatten)            (None, 6032)              0
_________________________________________________________________
dense (Dense)                (None, 120)               723960
_________________________________________________________________
activation_2 (Activation)    (None, 120)               0
_________________________________________________________________
dense_1 (Dense)              (None, 84)                1016
_________________________________________________________________
activation_3 (Activation)    (None, 84)                0
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 170
_________________________________________________________________
activation_4 (Activation)    (None, 2)                 0
=================================================================
Total params: 736,866
Trainable params: 736,866
Non-trainable params: 0
```

4. Conclusiones

La navegación autónoma es un problema muy difícil de resolver. Cada robot tiene sus peculiaridades y es complicado dar con un único algoritmo que resuelva el problema de la navegación de manera genérica. Lo que sí que podemos sacar en claro es que las soluciones "reactivas" siempre serán peores.

Además, en un sistema más complejo en el que hay una cadena de dependencias, pequeños cambios pueden resultar críticos. Como nos pasó en el caso de la navegación, que obteníamos mejores resultados con una cantidad menor de datos de entrenamiento.

Hemos aprendido que almacenar una representación del espacio real en el espectro digital tiene muchas complicaciones: memoria usada, precisión, puntos repetidos. En definitiva, la reconstrucción 3D es un problema complejo.

Otro aspecto a comentar es la combinación de diferentes métricas. Cuando nosotros pensábamos que el hecho de combinar odometría con imágenes rgb o de profundidad iba a suponer una mejora sustancial sobre el problema, este no devuelve los resultados esperados. Finalmente, conseguimos que resuelva el circuito con pequeñ

Finalmente, a la hora de tratar con nuevo software, es muy importante tener en cuenta las versiones compatibles. Debido a los numerosos problemas surgidos por las versiones de python, tensorflow o ros, se buscará utilizar siempre el software más actual y estable que podamos.

Referencias

[1] http://wiki.ros.org/Documentation

[2] http://wiki.ros.org/rosbag

[3] https://www.tensorflow.org/

[4] https://opencv.org/

[5] http://wiki.ros.org/stereo_image_proc

[6] http://wiki.ros.org/image_viewViewing_stereo_images