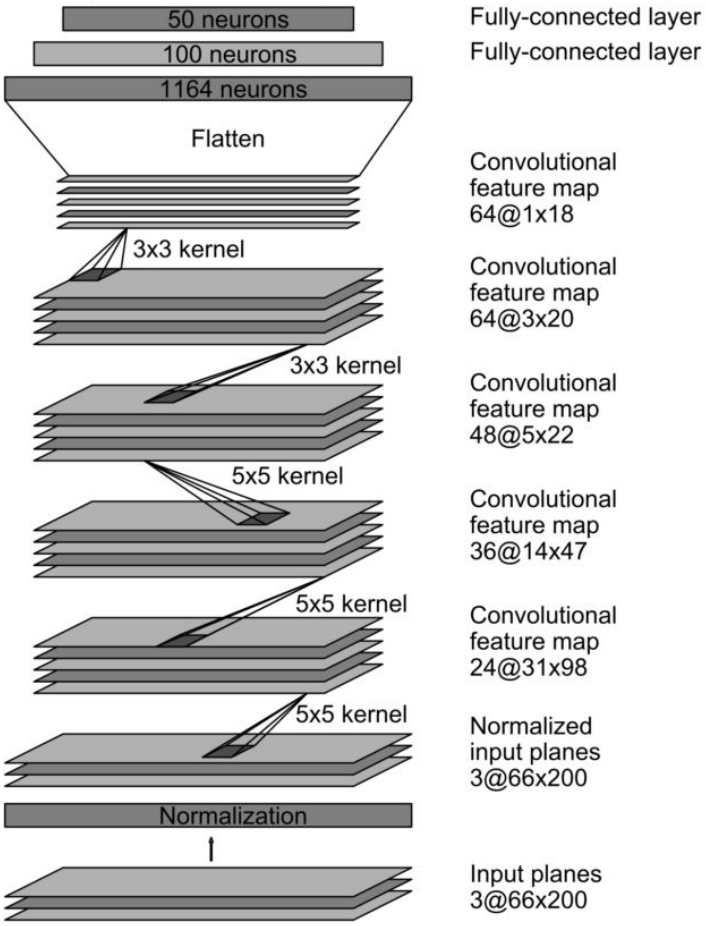
Vision Artificial Y Robotica

Práctica 1. Carrera de robots

1. Introducción



En este documento se va a explicar los pasos realizados para resolver el primer punto de la práctica. Navegación basa en deep learning. Para ello hemos usado una red neuronal diseñada por NVIDIA.

1.1 Estado tecnológico actual

La red convolucional PilotNet la presentó Nvidia e 25 Apr 2017, se trata de un red relativamente nueva y que la han usado profesionalmente para la conducción autónoma. Si la comparamos con la red que han usado algunos de los compañeros (como la mobileNet) es mucho más compleja. Esto hace que se trate de una red con más potencial aunque cueste bastante más tiempo entrenarla.

27 millones de conexiones frente a 3,538,984 conexiones que tiene mobilenetV2.

https://arxiv.org/abs/1704.07911

2. Navegación

2.1 Aproximación

2.2.1 Red Neuronal

Lo primero va a ser conocer en más profundidad la red neuronal. La red utilizada es conocida por PILOTNET y podemos encontrar mas informacion en el enlace https://arxiv.org/abs/1604.07316

La arquitectura de la cama la podemos ver en esta imagen que encontramos en el blog de NVIDIA. La red recibe de entrada una imagen RGB de 66 píxeles de ancho por 200 de largo. Está formada con una capa de normalización, 5 capas convolucionales y 3 capas *fully-connected*.
Además, en la implementación utilizada añaden algunas capas más en la salida.

Esta red neuronal a diferencia de alguna usada por nuestros compañeros no se encuentra en la base de *Keras*, asi que hemos realizado una búsqueda por internet y hemos visto algunos repositorios donde han implementado las distintas capas. Para evitar fallos y ahorrar tiempo de trabajo usaremos una de estas implementaciones.

Además estas implementaciones además vienen con un dataset y un modelo ya entrenado, lo cual es muy interesante para usar red ya entrenada y que nos sea más sencillo generar un modelo. Esta técnica se denomina *FineTuning*.

A continuación se explicará el proceso de cómo se han obtenido los datos y como se ha entrenado la red. Para generar el dataset se ha usado un controlador de consola donde se usa el joystick para mover el robot. De esta forma se va controlando el robot como si fuera un juego de carreras o de coches de una consola clásica. Tras realizar varias vueltas ya tenemos un dataset de miles de imágenes. Una de las mejores soluciones obtenidas se ha hecho con un dataset de 16.000 imágenes, sin embargo en algunos lugares determinados los giros realizados con el joystick no son lo suficientemente ‘contundentes’, es en esos lugares donde el robot falla, se dará más información adelante.

Una vez conseguido el dataset hay que generar el modelo de la red. Para ello hemos usado el servicio de Google Colab, ya que no disponemos de tarjetas gráficas con suficiente potencia para entrenar el modelo. Sin embargo usar google colab trae el problema de que es necesario subir el dataset y alguna dependencia más a internet para que este se la pueda descargar. Para ello hemos utilizado la plataforma DigitalOcean, que por ser estudiante tenemos credito suficiente para conseguir alojamiento.

Por último solo tenemos que ejecutar el script en python y esperar a que se entrene la red. En el apartado de ficheros encontraremos la salida obtenida y tenemos que descargarla. Una vez descargada se colocará dentro del nodo samplePy para que ejecutarle alimente la red.

A continuación vamos a hablar sobre el nodo samplePy que es el que se ocupa de predecir dada una imagen lo que debe girar el robot. Para ello samplePy carga los pesos del modelo ya entrenado y se subscribe a la cámara del robot, tras hacer la predicción publica la velocidad en el robot y este se mueve.

Adjunto un video donde se ven algunos de los resultados obtenidos.

https://drive.google.com/open?id=12ju9bbGLkNKl4bco8YPiWVi_EpRLLjbi

2.2.2 Joystick

Para generar el dataset se ha optado por usar un controlador de Xbox One para controlar el robot usando los joysticks. Esto se ha realizado así porque el proceso para mover el robot con el teclado es muy tedioso, sin embargo con el mando se hace mucho más cómodo incluso entretenido

A continuación se describe como se ha realizado.

Instalación de drivers en ubuntu

```
sudo apt-get install xboxdrv
sudo systemctl enable xboxdrv.service
sudo systemctl start xboxdrv.service
```

Ros

Nuestro simulador de robótica favorito incluye una series de paquetes que nos ayudaran a poder usar nuestro mando para mover el robot. Lo primero que tenemos que hacer es invocar al nodo joy

rosrun joy joy_node

Ahora tendremos un nodo al que podremos suscribirnos para obtener la información de los botones pulsados.

Por último y no menos importante debemos crear nuestro propio nodo que se suscriba al joy y publique en 'cmd_velocity' del robot el movimiento que queramos.

Sin embargo estamos de suerte ya que podemos encontrar en ros un nodo que podremos utilizar como base. Teleop_twist_joy De esta forma modificandolo podremos enviarle la información que queramos a nuestro robot. Este nodo se ha modificado para que también lea la cámara del robot y se escriba en el fichero de texto los datos.

2.2.3 Dataset

Formato del fichero

Para realizar el dataset vamos a seguir la idea de algunos de los ejemplos que encontramos en github, ya que es la versión más extendida y se ve claramente su funcionamiento. Nuestro dataset va a tener una columna de datos de entrada, donde se va a encontrar la ruta de la imagen. En la segunda y última columna se va a encontrar el valor numérico de lo que ha girado nuestro robot, valor que recordemos conseguimos gracias a teleop_twist_joy y nuestro controlador de la Xbox.

A continuación vemos un extracto del dataset, donde se entiende a la perfección lo explicado anteriormente.

```
951 ./VAR-DATASET/OpenCVImages/Image950.jpg -0.197963
952 ./VAR-DATASET/OpenCVImages/Image951.jpg -0.175106
953 ./VAR-DATASET/OpenCVImages/Image952.jpg -0.136878
954 ./VAR-DATASET/OpenCVImages/Image953.jpg -0.0951961
955 ./VAR-DATASET/OpenCVImages/Image954.jpg -0.0702994
956 ./VAR-DATASET/OpenCVImages/Image955.jpg -0.0324886
957 ./VAR-DATASET/OpenCVImages/Image956.jpg -0.000701123
958 ./VAR-DATASET/OpenCVImages/Image957.jpg -0
959 ./VAR-DATASET/OpenCVImages/Image958.jpg -0
960 ./VAR-DATASET/OpenCVImages/Image959.jpg -0
961 ./VAR-DATASET/OpenCVImages/Image960.jpg 0.00518252
962 ./VAR-DATASET/OpenCVImages/Image961.jpg 0.00791313
963 ./VAR-DATASET/OpenCVImages/Image962.jpg 0.00897324
964 ./VAR-DATASET/OpenCVImages/Image963.jpg 0.0100655
965 ./VAR-DATASET/OpenCVImages/Image964.jpg 0.0100655
```

¿Cómo se genera?
Para generar el dataset nos suscribimos al nodo hoy, y también a la vez al nodo de la cámara. Acto seguido cuando se recibe movimiento del joysticks procedemos a capturar la pantalla y guardar el valor para rotar.

Este método no es el más adecuado para crear el dataset ya que no existe una sincronía real entre los dos nodos, por lo que podría estar la imagen ligeramente desfasada respecto al valor de movimiento. Sin embargo parece que el dataset obtenido es lo suficientemente bueno, dado que conseguimos hacer que el robot aprenda.

Dataset definitivo

El dataset con el que hemos obtenido un rendimiento bastante bueno tiene un tamaño de 16000 imágenes. Simplemente nos hemos dedicado a dar vueltas al circuito. Esto ha sido un problema, ya que en los lugares más conflictivos el robot se choca, en otras pruebas hemos forzado esos giros de formas más brusca para que si tome las curvas.

2.2 Experimentación

TensorFlow y cuda en local

La primera idea principal fue instalar tensor flow de forma local en el portátil con tarjeta nvidia y entrenar la red con este. Así que se siguieron unos tutoriales para instalar TensorFlow, Cuda y demás requisitos. Se estuvo durante unos días entrenando la red, sin embargo el consumo energético, la temperatura y la velocidad fueron unos inconvenientes. Esto sumado a que debido a una instalación extra de una dependencia de Cuda se rompieron los drivers de la gráfica se decidió cambiar a Google Colab.

Durante el proceso de entrenamiento se han realizado las típicos pasos para encontrar el modelo más adecuado. Modificando algún parámetro interno de la red y las diferentes épocas. Sin embargo los mejores resultados se han obtenido dejado la mayoría de parámetro por defecto. Al final un buen resultado ha sido usar 60 épocas junto a un dataset de 16K. Sin embargo se han probado con dataset más pequeños y mas epocas (hasta 200!!). Algunos de estos intentos han sido para ver cómo se comportaba la red bajo overfitting. Que en general lo ha hecho bastante bien.

Para la recolección de datos para crear el dataset se realizó un primer intento usando el script que se proporciona en clase, Al pulsar una tecla y enter se envía un mensaje al simulador y a la vez se registra la velocidad de rotación y se guarda una captura. Sin embargo el problema principal que encontramos es que obtenemos un dataset con 3 etiquetas. 0, -0,75 y + 0,75 es por ello que al poco esta idea se descartó para buscar otra forma de controlar nuestro robot. Además, usar el teclado es muy lento y aburrido. Es incluso complejo controlarlo.

3. Imagen Stereo

En este apartado tenemos como objetivo la generación de una nube de puntos a partir de las imágenes de las cámaras traseras del turtlebot. Se tratan de dos Kinect V1, con una resolución de 640x480 píxeles.

Para ello hemos creado el nodo “stereo” que se ocupa de captar las imágenes de las cámaras traseras del turtlebot. Tuvimos que realizar modificaciones en el archivo `src/launch/kobuki_hexagons_kinect.urdf.xacro` : modificamos el apartado <frame_name> de cada cámara para poder distinguirlas programáticamente, y poder reutilizar el mismo callback para los flujos de imágenes de las cámaras.

3.1 Calibración de la cámara

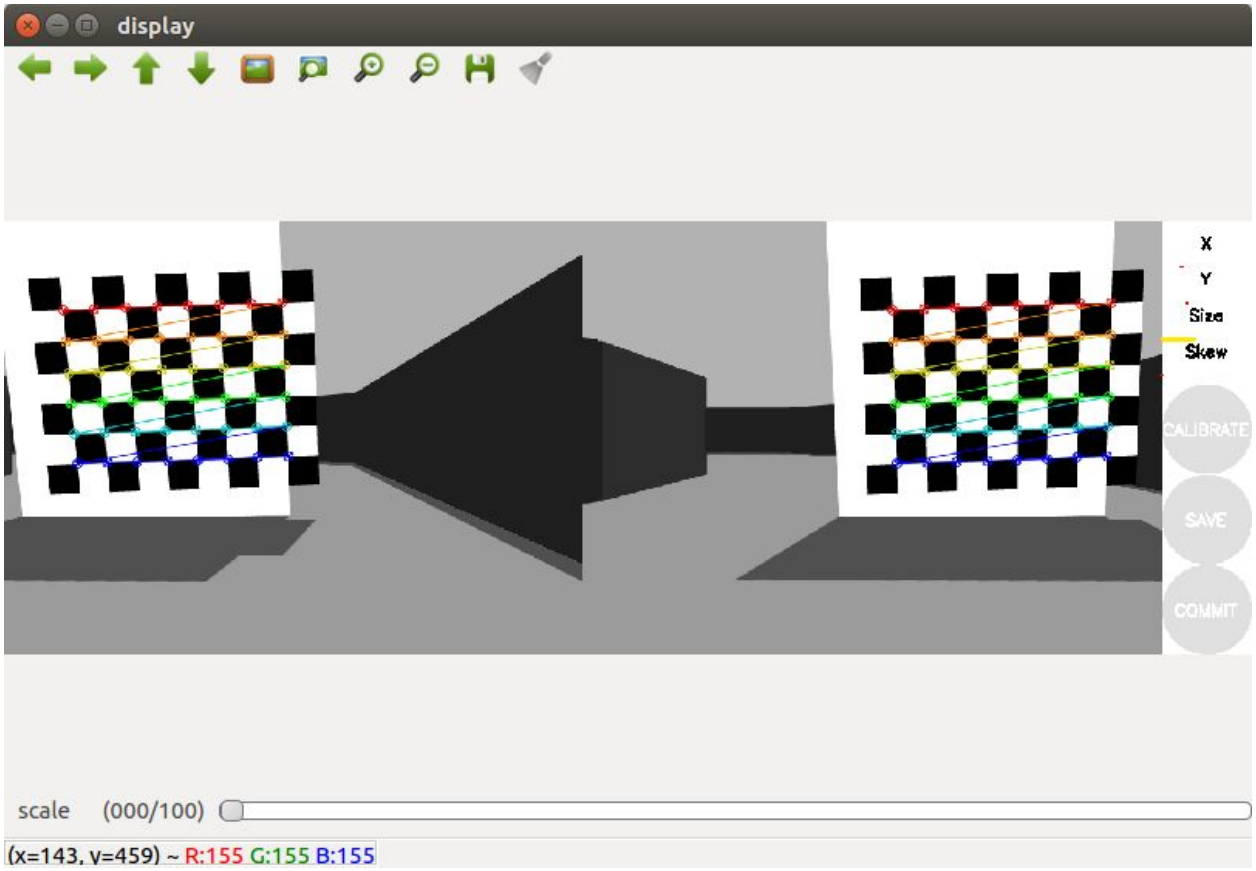
Uno de los los problemas más desafiantes de implementar el algoritmo ha sido el de encontrar los parámetros intrínsecos y extrínsecos de las cámaras. Probamos a buscar la información por internet, y a intentar calibrar las cámaras desde el propio Gazebo. Para ello utilizamos el paquete de ROS “camera_calibration”.

Con el comando siguiente ejecutamos la rutina de calibración:

```
rosrun camera_calibration cameracalibrator.py --approximate 0.1 --size 8x6
--square 0.108 right:=/robot1/trasera2/trasera2/rgb/image_raw
left:=/robot1/trasera1/trasera1/rgb/image_raw right_camera:=/robot1/trasera2/
left_camera:=/robot1/trasera1
```

Donde los parámetros left/right y left_camera/right_camera indican los tópicos a los que debe suscribirse la rutina para obtener el feed de imágenes de las cámaras izquierda y derecha respectivamente.
El parámetro `--approximate` permite a la rutina emparejar pares de imágenes derecha e izquierda aunque su timestamp no sea exactamente igual. Con el parámetro `--size` indicamos el tamaño del tablero que utilizaremos para calibrar las cámaras.

A continuación importamos un modelo de tablero en el simulador Gazebo y lo utilizamos para calibrar las cámaras:



Deberemos ir posicionando el tablero en distintas zonas de la cámara, cubriendo las zonas de los ejes x e y, rotando el tablero para ofrecer distintos ángulos a la cámara, y acerándolo y alejándolo para obtener muestras de distintos tamaños.

Una vez que obtengamos suficientes datos, se activará el botón ‘calibrate’ en el lateral derecho de la ventana, permitiendo iniciar el algoritmo de calibración y generando las matrices de parámetros intrínsecos de cada cámara (matriz de cámara, vector de distorsión, y matrices de proyección y rectificación (este último es característico de cámaras estéreo por lo que no lo necesitaremos).

3.2 Rectificación

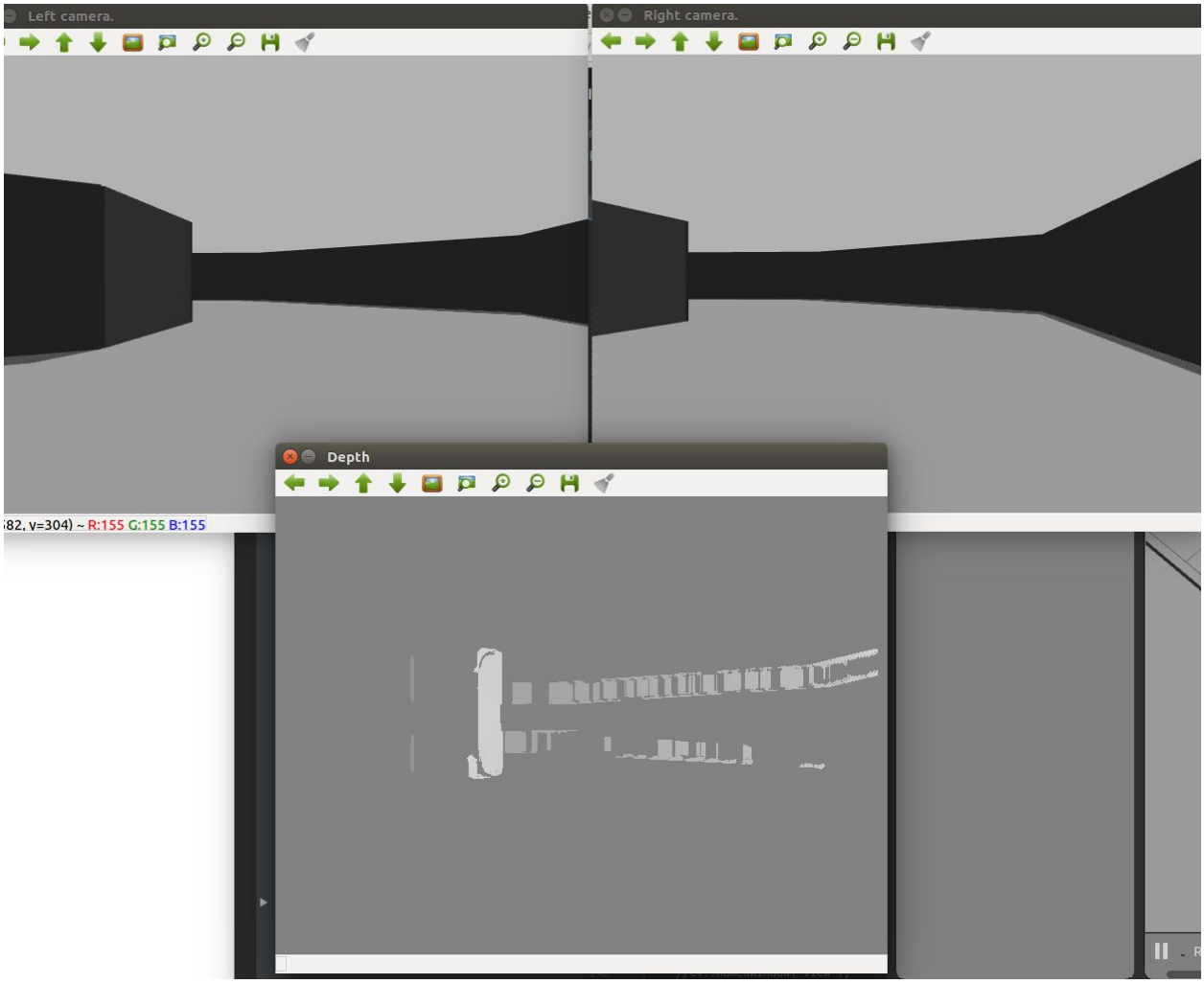
Una vez obtenidos los parámetros de de las cámaras podemos pasar a rectificar las imágenes para antes de aplicarles el algoritmo de matching. De esta forma se consiguen imágenes alineadas con la máxima correspondencia entre píxeles.

3.3 Stereo matching

Una vez rectificadas las imágenes, las recodificamos a escala de grises y ya están listas para ser procesadas. Aplicamos un algoritmo de matching que busca los píxeles correspondiente entre cada imagen. Podemos refinar el proceso cambiando algunos de los parámetros del algoritmo como la cantidad de

Con esto obtendremos la deseada nube de puntos, o imagen de profundidad. En ella observamos cómo los objetos más cercanos tienen un tono clarito mientras que los objetos más lejanos se colorean de un gris más oscuro.

Uno de los problemas que afectan al rendimiento del algoritmo es el coloreado del circuito: nos encontramos en un entorno que no se sale de una estrecha escala de grises, y con una iluminación simulada, lo que provoca que en muros de color uniforme, el algoritmo detecte exceso de similitudes y le sea imposible encontrar el píxel correspondiente en cada imagen.
Esta es la razón por la que en la nube de puntos destacan los bordes de las paredes y esquinas, ya que es donde encontramos más disparidad entre colores y donde el algoritmo es capaz de encontrar los pares de píxeles correspondientes de cada imagen.



Se puede apreciar como en la imagen de profundidad se detectan los bordes superiores e inferior del muro más el fondo. La sección del muro a la derecha va tomando un color más claro por su cercanía al turtlebot. Vemos cómo se detecta el cambio en el muro de la izquierda, y la diferencia en las esquinas y bordes, pero el algoritmo es incapaz de detectar los píxeles centrales del muro ya que interpreta demasiadas coincidencias por pixel.

4. Conclusiones

En la navegación la red neuronal PilotNet de nvidia ha demostrado ser una candidata excelente para la navegación de vehículos autónomos. Con relativas pocas épocas y un dataset sin equilibrar ya consigue moverse por el mapa con bastante soltura. Sin embargo debido a su tamaño es posible que no sea la mejor opción para esta práctica.

5. Herramientas Utilizadas

Software

- CLion 2018.3 - Plugin ROS Robotic Operating System.
-

Librerías y frameworks:

- Keras
- Tensorflow
- OpenCV

Lenguajes de programación:

- Python
- C / C++

Herramientas colaborativas varias (almacenamiento, versionado, compilación en línea)

- Google Colab
- Trello
- GitHub
- Servicio de almacenamiento de Digital Ocean.

6. Referencias

Calibración de cámara:
http://wiki.ros.org/camera_calibration

Información sobre algoritmos de estéreo en OpenCV:
https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

Redes neuronales y PilotNet:
https://towardsdatascience.com/convolutional-neural-network-to-steer-a-vehicle-inside-a-game-2aab41a5ef60
https://github.com/SullyChen/Autopilot-TensorFlow
https://keras.io/applications/
https://arxiv.org/abs/1704.07911

Xbox Controller:
https://www.maketecheasier.com/set-up-xbox-one-controller-ubuntu/
https://kyrofa.com/posts/your-first-robot-the-controller-3-5
https://github.com/ros-teleop/teleop_twist_joy