UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Deep Learning en microordenadores: aplicación con dispositivo Coral sobre Raspberry Pi

Trabajo Fin de Máster

Máster Universitario en Ingeniería Informática

AUTOR/A: Gavrishev Mesheryakov, Daniil

Tutor/a: Monserrat Aranda, Carlos

CURSO ACADÉMICO: 2021/2022

Resumen

Este documento recoge el procedimiento seguido para obtener un microordenador con capacidades de Inteligencia Artificial por medio de un acelerador USB. Se detallan los pasos seguidos así como los problemas surgidos durante el desarrollo, explicando las soluciones tomadas. Además se propone una solución real: crear un modelo para reconocer la carretera e indicar la dirección a un coche real y, un modelo que reconozca las señales de tráfico e indique acciones a realizar. Todo esto ejecutándose en el mismo microordenador con el acelerador USB.

Palabras clave: Inteligencia Artificial, microordenador, aprendizaje profundo, Coral, coche autónomo, visión artificial

Resumen

Aquest document arreplega el procediment seguit per a obtenir un microordinador amb capacitats d'Intel·ligència Artificial mitjançant un accelerador USB. Es detalla els passos seguits així com els problemes que han sorgit al llarg del desenvolupament del projecte, explicant les solucions que s'han pres. A més a més es proposa una solució real: crear un model per reconeixer la carretera i indicar la direcció a un cotxe real i, un model que reconega les senyals de tràfic i indicar accions a realitzar. Tot això executan-se amb el mateix microordinador amb el accelerador USB.

Paraules clau: Intel·ligència Artificial, microordinador, aprenentatge profund, Coral, cotxe autonom, visió artificial

Abstract

This document gathers the procedure followed to obtain a single-board computer with Artificial Intelligence capabilities by using an USB accelerator. It details the steps followed to obtain this result and the problems that took place as well as the explanation of how to solve them. Also it proposes a real solution: create a model capable of recognizing the road and indicate the direction to a real car, and a model capable of recognizing traffic signs and indicate different actions to follow. All this being ran in the same single-board computer with the USB accelerator.

Keywords : Artificial Intelligence, single-board computer, Deep Learning, Coral, self-driving car, computer vision

Tabla de contenidos

1. Introducción

La inteligencia artificial es un campo de la informática que cada día es más utilizado, estudiado y mejorado. A pesar de sus inicios estimados en la mitad del siglo pasado, es una ciencia relativamente nueva, consiguiendo muchos avances en muy poco tiempo en años recientes. Esto es posible gracias a que las tecnologías, *hardware*(1) y métodos usados han mejorado considerablemente, pero la premisa sigue siendo la misma: conseguir una máquina inteligente que se comporte como un ser humano.
Esta misma frase se puede interpretar de muchas formas: ¿qué es la inteligencia?, ¿cómo debe comportarse una máquina?, ¿qué implica ser un ser humano pensante?
Estas cuestiones filosóficas han sido debatidas durante años, cada persona teniendo su propia visión y expertos describiendo la inteligencia artificial como actuar racionalmente.

Actualmente la inteligencia artificial se puede encontrar en cualquier parte del mundo y en casi cualquier acción del día a día[1]. Desde sitios web, servicios o empresas, hasta bancos, mercado de valores. televisiones o un simple cable de luz que puede formar parte de un sistema monitorizado por inteligencia artificial para optimizar recursos y tiempo. La gran mayoría utilizan modelos entrenados previamente para realizar las diferentes tareas que tengan programadas, siendo estos entrenados con ordenadores con mucha potencia de cálculo para obtener modelos robustos y lo más precisos posibles.

Pero si la situación requiere usar inteligencia artificial en un lugar remoto o donde no se pueda disponer de un ordenador con potencia suficiente para utilizar el modelo entrenado, entonces se complican las cosas. Estas situaciones requieren de un ordenador o microordenador capaz de realizar tareas de inteligencia artificial sin muchos problemas y sin consumir mucha energía. Como por ejemplo, un coche autónomo que tenga que realizar cálculos y, además, realizar predicciones con un modelo en tiempo real, y lo más rápido posible para evitar cualquier tipo de error o accidente.

1.1 Motivación

La motivación principal para realizar este TFM(2) sobre Deep Learning y visión artificial es encontrar e implementar una solución para poder combinar un microordenador y tareas de visión artificial de una forma portátil y sin depender de elementos externos ni pesados. De esta forma, poder facilitar su uso y adquirir datos y manejarlos de una forma más rápida y fácil.

También quería extender mi conocimiento sobre inteligencia artificial y visión artificial ya que es un tema que me interesa, y que me sirva de base para seguir realizando más proyectos similares.

Otra motivación es la de conseguir crear un programa o sistema que se pueda implementar de forma relativamente sencilla en un vehículo y poder tener un coche real con cierta autonomía en la carretera. Además, este TFM puede servir de guía para otros usuarios tanto a la hora de crear su propio modelo de coche autónomo como para conseguir tener un microordenador con inteligencia artificial para sus proyectos de IoT[3], como ejemplo de uso, o similares.

1.2 Objetivos

1.2.1. Objetivo general

El objetivo principal del TFM es desarrollar una aplicación de *Deep Learning* sobre un microordenador, **Raspberry Pi**, utilizando **Python**, el acelerador USB[4] de **Coral AI** de **Google** y una cámara. Todo ello enfocado su uso al tránsito vial. Más concretamente, a la posibilidad de utilizar dicha aplicación para adaptar un coche real a un coche autónomo capaz de reconocer los carriles por los que se mueve y maniobrar dependiendo de las señales de tráfico y de los mismos carriles para no salirse de estos.

1.2.2 Objetivos específicos

- Poner en funcionamiento un microordenador y un acelerador USB y que se comuniquen ambos de forma correcta.

- Analizar la mejor forma de que haya una interacción fluida entre ambos dispositivos para poder realizar operaciones de *Deep Learning* sin necesidad de hardware ni ordenadores externos.

- Conseguir que un coche robot recorra un circuito sin interacción del usuario.

- Conseguir que el mismo coche robot recorra el mismo circuito después de "enseñarle" cómo y que reconozca señales de tráfico delante de él y reaccione a ellas.

- Adaptar los pasos realizados a una situación real: conseguir un sistema que indique qué hacer durante la conducción de un coche

1.3 Impacto esperado

Con este TFM se desea tener un impacto positivo en cuanto a mejorar los dispositivos IoT a la hora de adquirir y procesar información, ya que se utiliza *Edge Computing*, concepto que se explicará más adelante en el punto ***2.5 Alternativas***. Esto también puede aumentar la seguridad al no tener que acceder a la red obligatoriamente como ocurre con muchos dispositivos IoT que requieren tener acceso a la red para poder enviar y recibir datos.
A su vez, al ser un microordenador con un simple acelerador USB, su consumo es mucho más reducido que tener un ordenador de sobremesa realizando las mismas acciones. También está la posibilidad de conectar una batería al microordenador y poder transportarlo sin necesidad de ser apagado o tener pérdida de datos o información, caso que resulta ideal para ser utilizado durante trayectos en coche.

También se desea ofrecer una guía a cualquier usuario que quiera tener e implementar un microordenador con inteligencia artificial para realizar sus propios proyectos o continuar este TFM para tener su propio vehículo autónomo. De esta forma es posible encontrar todos los problemas que pueden surgir durante la instalación y configuración y una solución a dichos problemas en un mismo documento.

1.4 Estructura

En el punto ***1º. Introducción*** se introduce al lector la inteligencia artificial, los objetivos de este TFM y se exponen la motivación y los objetivos.

En el punto ***2º. Estado del arte*** se explica con más detalle el desarrollo de la inteligencia artificial desde sus inicios, con cada punto exponiendo las tecnologías a utilizar en este TFM.

En el punto ***3º. Análisis del problema*** se proponen soluciones, detallando la opción seleccionada, el análisis de seguridad y el plan de trabajo.

En el punto ***4º Diseño de la solución*** se indican y describen los diferentes componentes que se van a utilizar en el desarrollo.

En el punto ***5º. Desarrollo de la solución*** se relatan los pasos seguidos para conseguir el funcionamiento de un modelo *Deep Learning* en microordenador y desarrollar un sistema capaz de dirigir un coche real por la carretera, los problemas surgidos durante el curso del desarrollo y las acciones tomadas para solucionarlos y mejorar el funcionamiento.

En el punto ***6º. Resultados*** se exponen los resultados obtenidos, tanto del coche **PiCar** como de la aplicación real.

En el punto ***7º. Conclusiones*** se concluye analizando si se han cumplido los objetivos establecidos al principio y posibles mejoras a realizar.

En el punto ***8º. Bibliografía*** se facilita información relacionada con el TFM.

En el punto ***9º. Glosario*** se explican algunos términos y siglas utilizadas.

1.5 Convenciones

A continuación se explican las convenciones seguidas en la redacción de este TFM.

- *Cursiva*: palabras extranjeras
- **Negrita**: productos, marcas, nombres de *software* y nombres de *hardware*.
- `Courier New con fondo oscuro`: partes de código incluidos para dar más claridad o para destacar una parte importante, incluyendo rutas de directorios.
- *"Cursiva entrecomillado"*: citas literales, obras literarias y palabras literales extranjeras.
- ***Negrita y cursiva***: indicar palabras importantes que no se adapten a las convenciones ya establecidas y puntos del TFM

2. Estado del arte

2.1 Introducción

La inteligencia artificial (IA) o *Artificial intelligence* (*AI*)[1], es la ciencia e ingeniería de hacer máquinas inteligentes, especialmente programas inteligentes. Se relaciona con la tarea de utilizar computadores para entender la inteligencia humana, pero no es necesario que la IA se confine a sí misma a métodos que son biológicamente observables. Está definición ha sido presentada por John McCarthy en un artículo de 2004[2] donde John contestaba a diferentes preguntas relacionadas con la IA, siendo algunas de éstas más filosóficas como: *"¿Qué es la inteligencia?"* y *"Si la IA es una mala idea"*. También expone el comienzo del desarrollo e investigación de la IA, surgiendo después de la Segunda Guerra Mundial por personas independientes, pero muchos investigadores sugieren al matemático Alan Turing como el primero.

Alan Turing fué un matemático, criptoanalista e informático, entre otras áreas de conocimiento, britanico nacido en Londres en 1912[3], conocido por sus obras "*On Computable Numbers, with an Application to the Entscheidungsproblem"* y"*Turing's proof*" o Prueba de Turing, que consiste en tres pruebas que demuestran que "*Entscheidungsproblem"*[4], o problema de decisión escrito por David Hilbert en 1928, no tiene una solución concreta. En esta prueba, Turing sugiere la máquina de Turing, que consiste en la idea de una máquina abstracta que manipula símbolos en una cinta de acuerdo con una serie de reglas. Esta máquina opera con una cinta infinita dividida en celdas, las cuales pueden contener un único valor de un conjunto finito de símbolos llamado alfabeto, un cabezal que se posiciona sobre la celda y que lee únicamente la celda actual y escribe un símbolo y continúa o se para en función de las reglas iniciales y su estado actual. La idea es que está máquina puede realizar las tareas de cualquier otra máquina de computación. Una representación física de la idea se puede observar en la figura 1.

[1] Se referenciará como IA en varios puntos para mayor brevedad.

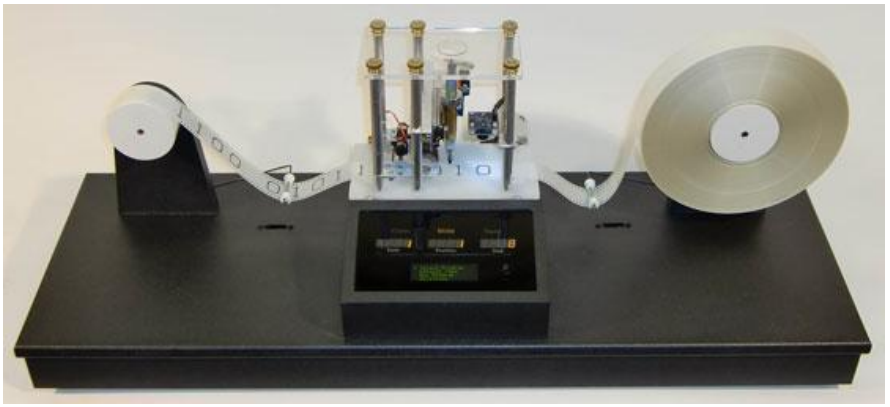


Figura 1: Representación física de la idea de una máquina de Turing, con cinta finita, en la idea sería infinita.

En el año 1950 cuando publicó "*Computing Machinery and Intelligence*"[5], Maquinaria computacional e Inteligencia, enfocado a la inteligencia artificial en el cual propone la pregunta de si las máquinas pueden pensar. Turing también sugiere crear un programa que imite la mente de un niño y, a partir de entonces, educarlo. Esto es debido a que la mente de un adulto se basa en varios factores, de los cuales destaca: el estado inicial de la mente, la educación recibida y experiencias vividas por la persona. Aquí Turing está haciendo referencia a *Machine Learning*, aprendizaje de máquinas, que es la rama de la IA dedicada a entender y crear métodos y técnicas que "aprenden" para mejorar su rendimiento en las tareas designadas a partir de información suministrada. Se puede entender que según Turing, la inteligencia es relativa, ya que diferentes personas pueden tener diferente inteligencia según sus experiencias y educación.
En un ejemplo reciente, el *chatbot*(5) de **Google**, **LaMDA**, tenía conciencia y era inteligente según un ingeniero que trabajaba con el programa. Esta afirmación fué desmentida por varios expertos en el tema, concluyeron que la conversación, a pesar de aparentar ser humana, no tenía signos de inteligencia[6]. Para el ingeniero, el programa pasó con éxito el test de Turing, mientras que para otros expertos no, concluyendo que la educación y experiencias de una persona son un factor de la inteligencia y, en este caso, para la prueba en sí.

En 1950 surgieron dos visiones para conseguir el objetivo de crear inteligencia artificial, una es la simbólica y la otra es la conexionista. La forma simbólica consiste en crear representaciones simbólicas del mundo, una representación que sea entendible por los humanos, utilizando para ello programación lógica(6), reglas de producto(7), redes semánticas(8) y *frames*(9). La forma conexionista pretende llegar a tener inteligencia artificial por medio del aprendizaje, utilizando para ello técnicas y algoritmos con *Artificial Neural Networks* (*ANN*) o redes neuronales artificiales.

Las redes neuronales artificiales son una colección de nodos interconectados entre sí, aparentando ser neuronas en un cerebro humano. Cada nodo recibe una señal que debe procesar y enviar a otro nodo si es necesario, estas señales son números reales. Tanto los

nodos como las conexiones entre ellos, normalmente tienen unos pesos que se ajustan durante el proceso de aprendizaje.
Estas redes están basadas en un artículo realizado por Warren S. McCulloch y Walter Pitts, "*A logical calculus of the ideas immanent in nervous activity*"[7], en el cual se buscaba entender cómo un cerebro humano produce patrones e ideas complejas por medio de las neuronas. De este artículo surgió la idea de comparar las neuronas con 0 y 1, falso y verdadero.
El funcionamiento de estas redes requiere de una entrada a la cual se asignan unos pesos en función de la importancia que tenga cada variable de la entrada. Los valores de la entrada entonces se multiplican por sus respectivos pesos y se suman. A continuación, estos valores pasan por una función de activación, activándose el nodo si el resultado es superior a un determinado valor, pasando el resultado a la siguiente capa como entrada del siguiente nodo. En la figura 2 se puede observar como es la estructura que tienen estas redes.

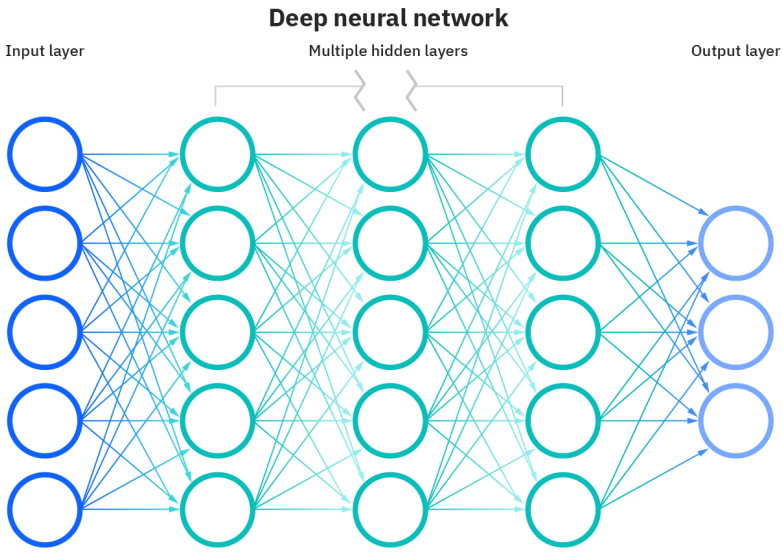


Figura 2: Estructura de capas de una red neuronal artificial

En 1958, Frank Rosenblatt inventó el perceptrón[8], la primera red neuronal artificial, consiguiendo que un ordenador pueda diferenciar entre cartas marcadas del lado derecho o del lado izquierdo. Unos años más tarde, en 1965, se crearon las primeras redes funcionales con muchas capas, publicado por el matemático sovietico y ukraniano Alexey Ivakhnenko.

Hasta 1974 se produjeron grandes avances en el campo de la IA, momento desde el cual, la investigación se ralentizó al encontrarse con tareas demasiado complicadas de solucionar. Destacando a Paul Werbos, que utilizó un algoritmo de propagación hacia atrás en redes neuronales[9], utilizando para ello el método de diferenciación automática, publicado en 1970 por el matemático finlandes Seppo Linnainmaa. En 1980, con el éxito de los "sistemas experto"', que consiste en un programa de IA que simulaba a un

experto de un campo en particular y sus decisiones a tomar, el interés por la IA volvió a crecer, hasta 1987 cuando colapsó el mercado de las **máquinas Lisp**[10]. Se empezó a cuestionar si el enfoque simbólico era suficiente para conseguir una imitación de los procesos cognitivos de un humano, emergiendo de nuevo el interés por el enfoque conexionista y las redes neuronales.
A pesar de haber existido programas y algoritmos con un enfoque conexionista en años anteriores, el término de *Deep Learning* se introdujo por primera vez por esta fecha.
A partir de aquí, se fueron mejorando las técnicas y métodos utilizados, en reconocimiento de objetos 3D, predicciones del mercado de valores e incluso coches semi-autónomos.

En 2012, Andrew Ng y Jeff Dean crearon una red neuronal que podía reconocer objetos complejos solamente con imágenes sin etiquetar[10]. Esto se consiguió gracias a la mejora en GPUs y ordenadores más potentes, que permitieron mejorar el aprendizaje no supervisado al poder tener redes neuronales más grandes. Estas redes neuronales mejoraron hasta el punto de alcanzar precisiones parecidas a la de los humanos.

2.2 Deep Learning

Deep Learning es un campo de *machine learning* o aprendizaje de máquina, que a su vez, esta es una rama de la inteligencia artificial. La diferencia principal entre ambos conceptos es en la forma en la que aprenden: mientras que *machine learning* requiere de más interacción humana para funcionar correctamente, *deep learning* puede utilizar grandes cantidades de información sin procesar, como textos o imágenes, con muy poca interacción humana. *Machine learning* requiere de datos estructurados y definidos previamente por una persona que son los que se suministran al programa. *Deep learning* no requiere de este proceso ya que es capaz de procesar información no estructurada y "aprender"' durante el proceso de propagación. Por ejemplo, si queremos distinguir entre varios tipos de coches, con *machine learning* tenemos que indicarle qué tipo de coche es en todas las imágenes que pasamos al programa, mientras que con *deep learning*, el programa aprende qué rasgos son los más importantes para definir cada tipo de coche.
Existen tres tipos de aprendizaje:

- Aprendizaje supervisado: en este proceso se proporcionan ejemplos de entrada con sus etiquetas[11] y la salida que se desea tener, el programa entonces aprende reglas, que son funciones, que el programa ajusta hasta encontrar la función que mejor ajusta los datos de entrada a la salida.

  Los problemas de aprendizaje supervisado se pueden diferenciar en problemas de clasificación y regresión. En los problemas de clasificación se buscan categorías en su salida, por ejemplo, diferenciar entre tipos de coches o colores,

mientras que en los problemas de regresión se busca conseguir valores reales en su salida buscando alguna relación entre variables dependientes e independientes, como por ejemplo proyecciones de venta de una empresa o distancia entre objetos.

- Aprendizaje no supervisado: en este proceso el programa debe aprender y encontrar estructuras de datos y rasgos en los datos de entrada, sin que se le aporte ningún tipo de información ni ejemplos, para que de esta forma, aprender de esos mismos datos y ofrecer otros datos de salida relevantes, por lo que no hay una respuesta correcta o incorrecta. Esto se consigue intentando imitar los datos de entrada y utilizando el error en la salida para corregirse a sí mismo.
  Se pueden diferenciar tres tipos, *clustering* o agrupación, asociación y reducción de la dimensión.
  Como su nombre indica, con la agrupación se intenta encontrar grupos con los datos de entrada identificando sus similitudes y diferencias, por ejemplo identificar grupos de clientes que compran ciertos productos.
  Con la asociación se busca encontrar asociaciones entre datos, por ejemplo en recomendaciones de videos o productos, cuando se realizan búsquedas, se mira o se compra un tipo de video o producto, se recomienda otro con características similares.
  La reducción de la dimensión es una técnica que busca reducir el número de datos de entrada si se proporcionan datos de entrada con un número elevado de variables, manteniendo las propiedades de los datos de entrada. Por ejemplo, para eliminar ruido en una imágen para mejorar la calidad de la misma.

  Existe un segundo tipo de aprendizaje no supervisado llamado semi-supervisado, en el cual se le proporciona una pequeña cantidad de información inicial y el programa aprende las diferentes estructuras y rasgos teniendo en cuenta esos datos de entrada.

- Aprendizaje de refuerzo: en este proceso, durante el proceso de aprendizaje, el usuario le proporciona una respuesta a las acciones realizadas por el programa hasta que alcanza el objetivo propuesto. De esta forma los errores cometidos por el programa se pueden supervisar y corregir. Se utiliza un sistema de recompensa y castigo para cada acción realizada por el programa, donde este intenta maximizar las acciones de recompensa. Como ejemplo podemos encontrar el caso de **AlphaGo Zero**, que fue el primer programa *Deep Learning* en vencer al campeón del mundo Lee Sedol al juego **Go**[12][14](figura 3).

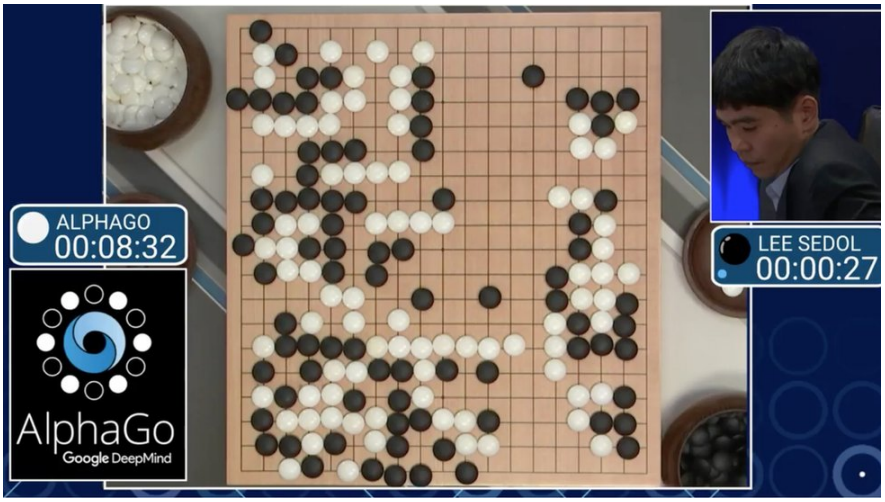


Figura 3: AlphaGo en la partida contra Lee Sedol

Las redes neuronales utilizadas pretenden simular un cerebro humano, con capas de nodos interconectados que trabajan con el resultado de la capa anterior para refinar y optimizar una predicción o categorización. Las capas donde se introducen los datos y donde se recoge el resultado se denominan capas visibles, y es, en está última capa, donde la predicción o categorización final se realiza.
También es posible encontrar propagación hacia atrás, donde se utilizan algoritmos para calcular los errores de las predicciones de las capas internas, ajustando los pesos y la parcialidad moviéndose hacia atrás en las capas. Ambas propagaciones se utilizan para mejorar los resultados y reducir los errores.
Al igual que *deep learning*, *machine learning* también utiliza redes neuronales, pero el concepto de *deep* se refiere a utilizar más de tres capas.

2.2.1 *Convolutional neural network (CNN)* o red neuronal convolucional

Muchos de los modelos actuales están basados en esta red neuronal, estos se utilizan en tareas de clasificación, reconocimiento y visión artificial. CNN es una versión regularizada de *multilayer perceptron (MLP)* o red perceptrón multicapa, que son redes multicapa donde cada nodo de las capas está unido a todos los nodos de la capa siguiente. Estas son propensas al *overfitting* o sobreajuste[13] por lo que es recomendable regularizar los datos. También, para evitar sobreajustes, se penalizan ciertos parámetros durante el proceso de entrenamiento o se ignoran ciertas conexiones entre nodos.
La diferencia con CNN es la forma de regularizar los datos, ya que ésta construye patrones a partir de más simples y pequeños patrones donde las primeras capas se centran en características más pequeñas como colores y siluetas. A medida que los datos pasan por las capas, se aumenta la complejidad y se identifican mayores elementos y formas del objeto hasta que se identifica correctamente.
CNN está formada por tres tipos principales de capas: capa convolucional, capa de agrupación y capa completamente conectada.

- La capa convolucional es la primera de todas, puede ser continuada por otras capas convolucionales o por capas de agrupación. Requiere de datos de entrada, un filtro y un mapa de características. Si por ejemplo se quiere clasificar una imágen, se le proporciona una matriz de píxeles en 3D[14] de la imágen como datos de entrada, teniendo su anchura, altura y profundidad correspondiente al RGB[15] de la imágen, además se tendrá un detector de rasgos, también llamado *kernel* o filtro, que se moverá por la imágen buscando si el rasgo está presente. El detector de rasgos es un vector bidimensional de pesos, típicamente de tamaño 3x3 pero pueden cambiar, que representa parte de la imágen, este filtro se aplica a una área de la imágen, llamado campo receptivo, produciendo un valor con el producto de los pesos del filtro y el área de la imágen que se incluye en un vector de salida. El filtro se mueve un *stride*, que es el número de píxeles que se debe de mover el *kernel* sobre la matriz de entrada, y se repite el proceso hasta que el filtro recorre toda la imágen. El vector final es el mapa de rasgos (figura 4).
  La funcionalidad del filtro es muy parecida a lo que realizaría un filtro en los diferentes tipos de transformaciones de una imágen realizadas en procesamiento de imágen.

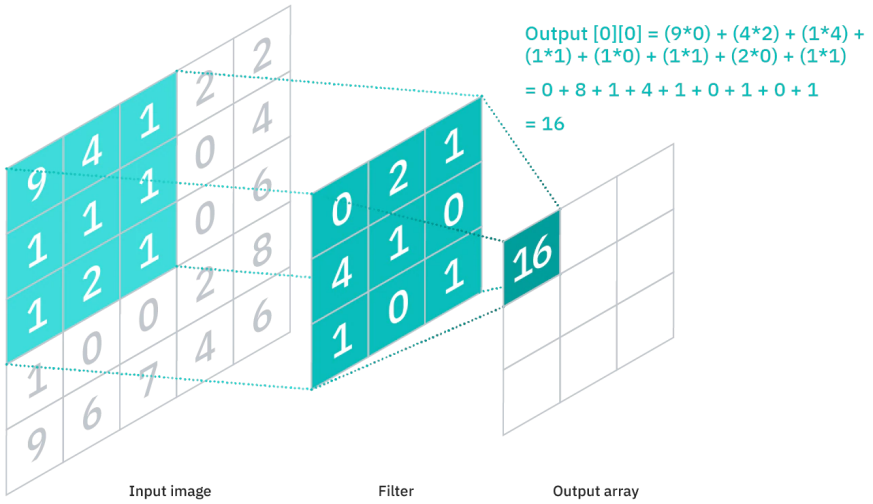


Figura 4: Funcionalidad de un filtro

Los pesos del filtro son invariables a medida que este avanza por la imágen, pero son ajustables por medio de la propagación hacia atrás y gradiente descendiente[16]. Se requiere de tres parámetros para modificar el tamaño del vector final que se deben configurar antes de empezar el entrenamiento de la red neuronal, estos son: el número de filtros que producirían diferentes mapas de rasgos y definirían la profundidad del vector final; el *stride*; y *zero-padding* o relleno cero, que pone los elementos que se encuentren fuera de la matriz de entrada a cero y se usa cuando los filtros no se ajustan correctamente a la imágen de entrada. También se puede tener tres tipos de relleno: *valid padding* o relleno válido donde si las dimensiones no son iguales, se ignora los últimos

movimientos del detector de filtros; *same padding* o mismo relleno que se asegura que el vector de salida tenga el mismo tamaño que la capa de entrada; y *full padding* o relleno completo que añade ceros en los bordes del vector de salida.
Una vez el filtro haya pasado por el campo receptivo de la imágen, se aplica una transformación *ReLU* (figura 5) al mapa de características, esto es una función de activación[17] que obtiene la parte positiva de los argumentos proporcionados. A día de hoy, es de las funciones más utilizadas en *deep learning*.

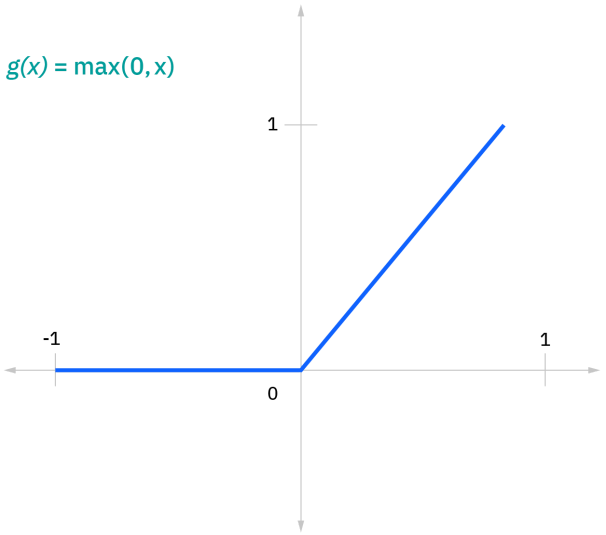


Figura 5: Función de activación *ReLu*

- Las capas de agrupación reducen el número de parámetros en la entrada pasando un filtro por todos los datos de entrada, al igual que la capa convolucional. Se aplica una función de agregación a los valores del campo receptivo y produce un vector de salida. Se distinguen entre *max pooling* o agrupación máxima, donde se seleccionan los píxeles con el máximo valor; y *average pooling* o agrupación media, donde se calcula la media de los píxeles del campo receptivo.

- La capa completamente conectada es la última de todas y solo puede haber una ya que aquí es donde se realiza la clasificación en función de los rasgos extraídos de las capas previas y sus filtros. Normalmente, el resultado será un valor entre 0 y 1 en función de la certeza que se tenga.

2.2.2 *Recurrent neural network (RNN)* o red neuronal recurrente

Esta red utiliza datos secuenciales o datos de una serie temporal, utilizados en problemas de traducción de lenguaje, procesamiento de lenguaje natural, reconocimiento de voz y descripción de imágenes. A diferencia de las otras redes

neuronales, esta red tiene "memoria", ya que ajusta y cambia su entrada y salida en función de la información obtenida de entradas anteriores.
En RNN se comparten parámetros entre capas, donde en otras redes el uso de estos pesos son exclusivos para cada capa, pero también se ajustan con el proceso de propagación hacía atrás para facilitar el entrenamiento. Esta propagación es diferente ya que se propaga sobre el tiempo, normalmente calculando el error entre capas de entrada y salida durante el entrenamiento, pero ahora se suman los errores en cada momento del tiempo, ya que los parámetros son compartidos por las capas.
Al utilizar este tipo de propagación, la red neuronal recurrente, y en todos los modelos de *Deep Learning*, ocurren varios problemas, *exploding gradient* o explosión de gradiente y *vanishing gradient* o desvanecimiento de gradiente, la gradiente en este caso siendo la derivada de una función que tiene más de una variable como entrada. El primer problema hace referencia a la gradiente siendo muy grande, aumentando además los pesos hasta que no se puedan representar; mientras que el segundo problema es parecido pero la gradiente es demasiado pequeña, y continúa reduciéndose, además de actualizar los pesos, hasta que el resultado es casi cero, momento en el cual, la red deja de 'aprender'.

Long short-term memory (*LSTM*) o memoria a corto largo plazo, es una de las arquitecturas RNN más utilizadas. Diseñada por Sepp Hochreiter y Juergen Schmidhuber[15] para intentar encontrar una solución al problema del desvanecimiento de la gradiente, proponen una solución al problema de las dependencias de largo plazo, esto quiere decir que si la capa actual recibe datos que afecten a su predicción y esos datos no son recientes, es probable que la red no pueda predecir correctamente su estado actual. Para esto se tiene "células" en las capas intermedias de la red que están formadas por tres puertas: de entrada, de salida y de olvido. La información va cambiando a medida que el proceso de aprendizaje transcurre, por lo que estas puertas deciden qué información es relevante y, por tanto, se debe mantener, y qué información es prescindible y, por tanto, eliminar. Las puertas suelen usar la función de activación sigmoidal (figura 6), esta función transforma la información en un valor entre cero y uno. Si un valor es multiplicado por cero, el resultado es cero y se olvida.

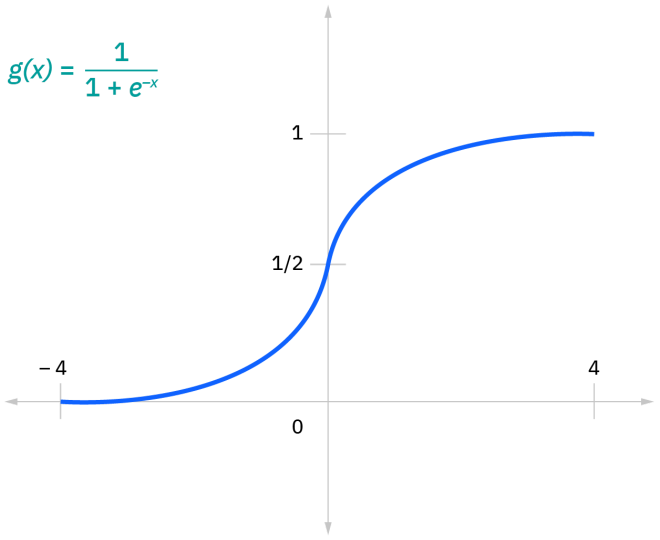


Figura 6: Función de activación sigmoidal

Gated recurrent units (*GRUs*) o unidades recurrentes cerradas, es una arquitectura similar a *LSTM*, diseñada por Kyunghyun Cho[16] para resolver el mismo problema de memoria de corto plazo, pero está vez se utilizan estados ocultos en vez de células y dos puertas en vez de tres: una puerta de reinicio y la otra de actualización, con el mismo fin de controlar qué mantener y qué olvidar. La puerta de actualización es la que indica qué mantener y qué olvidar, como la puerta de olvido en *LSTM*, y la puerta de reinicio indica cuánta información olvidar.

2.3 Visión artificial por computador

La visión artificial es un campo de la IA dedicado a obtener información de imágenes y videos de la misma forma que lo haría un ser humano. A diferencia de la visión de un ser humano, un programa de visión artificial tiene la desventaja de no tener conocimiento previo de objetos, formas o colores, por lo que es necesario un entrenamiento previo. El objetivo en este caso sería tener una entrada de una imágen simple, de un vídeo o un objeto en 3D, por medio de una cámara o que se le proporciona, convertirlo en un vector que pueda ser entendible por el programa, procesar dicha imágen en función de los algoritmos utilizados y analizar el resultado para entender qué clase de imágen es y qué contiene.

El desarrollo de la visión artificial comenzó alrededor de 1960 en universidades con la introducción de la IA como campo de estudio. Por las mismas fechas, neuropsicólogos descubrieron que el procesamiento de imágenes en un ser vivo comienza por reconocer las siluetas de una forma. También se desarrolló tecnología capaz de recoger y digitalizar imágenes del mundo real, y en 1963, permitía transformar imágenes 2D en 3D.

A partir de 1970 se desarrollaron muchos de los algoritmos utilizados hoy en día, como extracción de siluetas, modelado poliédrico y representación de objetos como formas más pequeñas entre otros. Otra tecnología desarrollada fue el reconocimiento de caracteres ópticos e inteligentes, que permitía reconocer textos y textos escritos a mano, lo cual se utiliza sobre todo en traducciones, reconocimiento de matrículas o procesamiento de documentos.

En 1982, Kunihiko Fukushima[17], desarrolla una red neuronal de células capaz de reconocer patrones en imágenes utilizando capas convolucionales.

En los próximos años se dedicó esfuerzo en análisis matemáticos, comprender mejor el funcionamiento visual de las cámaras, desarrollo de métodos en 3D, segmentación de imágenes y, por primera vez, se usaron técnicas estadísticas para reconocer caras en imágenes, siendo en 2001 cuando aparecieron las primeras aplicaciones de reconocimiento facial en tiempo real[18].

A partir del año 2000 se empezó a centrar en el reconocimiento de formas y en la estandarización de cómo se deben etiquetar y anotar los datos visuales. En 2010 surgió **ImageNet**, una biblioteca con millones de imágenes etiquetadas, dando pie a desarrollos de modelos con *deep learning*, mejorando significativamente los resultados obtenidos reduciendo la tasa de error casi al mínimo.

Hoy en día, *deep learning* está presente en casi todas las aplicaciones de visión artificial, desde coches autónomos[19] (figura 7) hasta detección de cánceres en medicina[20] (figura 8). En este tipo de aplicaciones, gran parte del esfuerzo se concentra en obtener el mayor número de imágenes y etiquetarlas correctamente.

Figura 7: Coche autónomo sin conductor

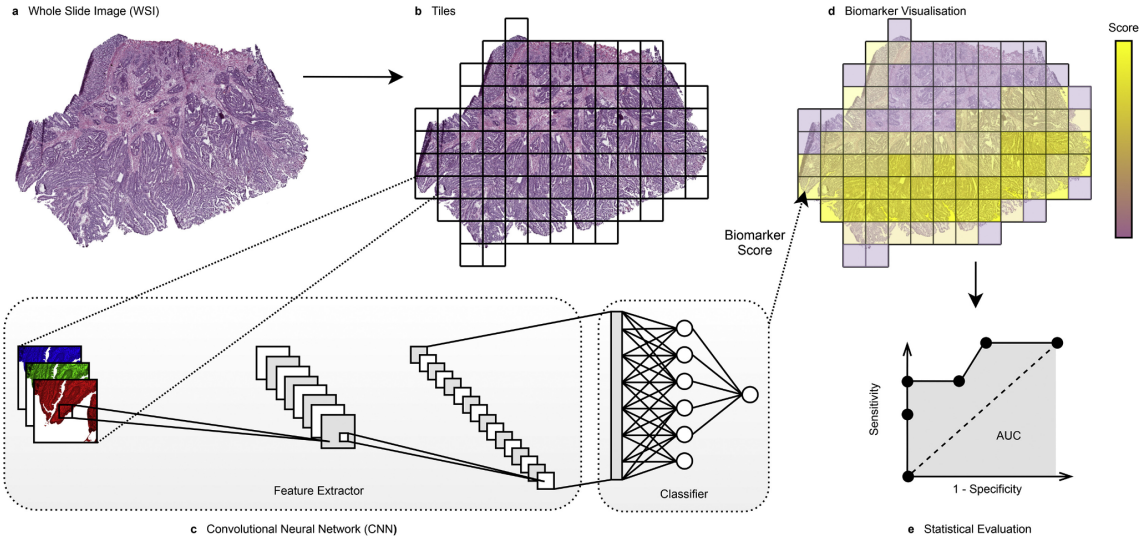


Figura 8: Clasificación de cáncer utilizando *deep learning*

Para que un programa de visión artificial funcione correctamente, se debe proporcionar bastante información de la tarea que se quiere llevar a cabo ya que se debe de poder reconocer ciertos patrones correspondientes a cada objeto. Por ejemplo, para detectar señales de tráfico, es necesario aportar al programa imágenes de señales de tráfico con diferentes fuentes de luz, deformaciones y tamaños para reconocer patrones que se tengan en común y que sea lo más preciso posible. Aquí es donde entra el *deep learning* y las redes neuronales convolucionales. Al igual que una persona, y con el descubrimiento de 1960 mencionado anteriormente, con la red neuronal convolucional primeramente se reconocer siluetas y formas sencillas y luego se profundiza en los detalles de la imágen hasta reconocer el objeto final. También es posible utilizar redes neuronales recurrentes para analizar imágenes de un vídeo y decidir si tienen relación entre ellos.

2.4 Arquitecturas de modelos y *frameworks*

Con los avances en visión artificial, actualmente existen muchas tareas que se pueden realizar de diferentes formas. Entre las más comunes podemos encontrar tareas de clasificación, detección y/o localización de objetos en imágenes 2D y 3D, reconocimiento de caras y posiciones de personas, segmentación y transformaciones faciales. Para cada tarea se han desarrollado muchas y diferentes arquitecturas de modelos a utilizar, algunos más rápidos, otros más precisos y otros más ligeros[21].
Un modelo es generalmente un fichero con datos del entrenamiento que se llevó a cabo junto con las capas por las que deben pasar los nuevos datos para ser analizados. La salida consiste en una etiqueta del objeto detectado y/o clasificado y la confianza de la predicción. Una vez el modelo es entrenado, la estructura interna no se ve afectada por nueva información, a menos que el modelo esté diseñado para realizar estos cambios. Por lo general, se debe de volver a entrenar un modelo si se quiere mejorar su precisión o eficiencia.

Para utilizar un modelo es necesario tener un *framework* donde poder cargar los datos de un modelo entrenado y poder realizar llamadas a funciones que se encargan de realizar las operaciones necesarias para predecir y calcular los resultados.

Entre los *frameworks* más utilizados destacan: **TensorFlow**, uno de los *frameworks* más populares, *open-source*(18), completo, con posibilidad de trabajar en PC, móvil, web y en la nube, y dedicado a *machine learning*; **Keras**[22], simple y flexible ahora es parte de **TensorFlow**[23] por lo que comparten características; **Caffe**[24], un *framework* creado pensando en la velocidad, flexibilidad y expresiones en vez de código en mente; **PyTorch**[25], un *framework open-source* centrado en *deep learning*; **JAX**[26], que combina **Autograd** y **XLA**; o **MXNet**[27], un *framework* de **Apache**(19) para *deep learning* que es flexible y eficiente.

Cuando se utilicen modelos en ordenador, la elección es amplia y variada dependiendo de las necesidades. Pero si lo que buscamos es utilizar un modelo en móvil o microordenadores como es el caso de **Raspberry**, es necesario tener en cuenta una serie de factores para poder desplegarlo. Uno de estos factores es comprobar que tanto el *framework* como el modelo no ocupen una cantidad de espacio significativa. Otro factor es que el modelo esté optimizado o, al menos, permita utilizar GPU(20) o TPU(21), ya que la potencia de la CPU(22) de un móvil o microordenador es limitada. Otros dos factores importantes, que además afecta al modelo utilizado en este TFM, es la posibilidad de poder cuantizar los valores del modelo, esto significa cambiar los valores en coma flotante a valores enteros para compactar el modelo y poder utilizarse en *hardware* más restringido en cuanto a computación[28], y el segundo siendo el tipo de operaciones utilizadas por el modelo, como por ejemplo, la operación **ReLU** no está soportada por la **Coral AI**.
La elección de un *framework* correcto también es importante ya que algunos modelos dependen de un *framework* u otro. En este caso, se utilizará **TensorFlow** porque uno de los modelos recomendados para **Coral AI** y el que se ha utilizado, es **EfficientDetLite**. Otros modelos como el popular **YOLO** (*You Only Look Once*, solo miras una vez) se puede ejecutar en diferentes *frameworks* como **PyTorch**, **ONNX** o **Darknet** entre otros.

2.5 Alternativas

Para hacer programas de inteligencia artificial y/o *machine learning* funcionales utilizando modelos, generalmente se requiere gran capacidad computacional. Mientras que un ordenador de sobremesa no tendría ningún tipo de problema, utilizar microordenadores como es el caso de **Raspberry Pi** o **Arduino** no es recomendable, ya que la CPU de estas placas no es suficientemente potente.
Para remediar esto y tener un sistema funcional de tamaño reducido, es necesario encontrar alternativas de poco coste y que además puedan ejecutar modelos con bastante precisión.

Una posible solución es enviar los datos utilizando la red, pero esto no es recomendable y, en algunos casos, incluso imposible. Esto es debido a que hay posibilidades de que se utilicen los microordenadores en lugares remotos, con poco o nulo acceso a la red o que la respuesta tenga que ser instantánea. Otro factor relacionado es la latencia con la que se reciben los datos.
Por otro lado nos encontramos con la velocidad a la que la CPU puede realizar cálculos, siendo bastante significativos. Como se explicó anteriormente, los modelos en general utilizan vectores para realizar cálculos, las CPUs no están diseñadas para ese tipo de cálculos, pueden realizar los cálculos pero no están especializados como lo son las GPUs que destacan por realizar cálculos vectoriales con mucha rapidez[29]. Las GPUs son elementos *hardware* compuestos por cientos de núcleos que además tienen una gran capacidad de paralelismo. Por tanto, la CPU puede delegar cálculos, en este caso vectoriales, para aprovechar este paralelismo y obtener resultados mucho más rápido que si los ejecuta por sí mismo.
El tamaño podría ser otro factor, pero en menor medida si los comparamos con los factores mencionados anteriormente. Se podría intentar conectar una GPU a un microordenador, pero actualmente no está soportado por los propios desarrolladores del *hardware*. Existen formas de conectar ciertas GPUs antiguas, pero esas formas no forman parte de lo "estándar" a la hora de la utilización del *hardware*.

Por tanto, si se quiere tener un sistema funcional, rápido a la hora de realizar cálculos, asequible, de tamaño reducido e independiente, en el sentido de no necesitar *hardware* externo para funcionar como puede ser un dispositivo IoT que requiere de *hardware* externo para realizar cálculos, *edge computing* es una solución muy acertada.
Edge computing[30][31][32] es la capacidad de tener computación, almacenamiento y recolección de datos cerca o en el mismo lugar donde se recogen dichos datos. Un ejemplo sería tener un coche autónomo que capte los datos de la carretera y señales, realice los cálculos pertinentes y los guarde para su uso posterior.
Edge devices[33] son dispositivos que hacen uso de esta arquitectura y procesan los datos en el sitio, para luego transmitirlos o guardarlos. Actualmente existe un número limitado de este tipo de dispositivos:

- Las *Tpu's* (*Tensor Processing Unit*) que han sido desarrolladas por **Google** para ser utilizadas con *machine learning*. Actualmente solo son accesibles por medio de su plataforma en la nube utilizando **Colab**, pero en 2019 lanzaron varios productos para *edge computing* que incorporan el chip **Edge TPU**. Este tiene la función de una TPU pero con menos potencia. Los productos (figura 9) se distribuyen bajo la marca **Coral**, donde se incluye el acelerador USB utilizado en este TFM. Es compatible con sistemas operativos basados en **Debian** y **ARM64**, como la **Raspberry Pi**, y únicamente soporta operaciones de 8-bit, por lo que es necesario cuantizar los datos de entrenamiento durante el mismo proceso o posteriormente con una herramienta.

Figura 9: Dispositivos de **Coral AI**

- La compañía conocida por sus GPUs, **Nvidia**, también ha lanzado productos de *edge computing*, que incluyen la familia **Jetson**: **Nano, TX2, Xavier NX, AGX Xavier, Orin NX** y **AGX Orin** (figura 10). Utilizan su propia distribución de **Linux** llamada **Linux for Tegra**. Es posible adquirir estos productos por separado, y tener únicamente un elemento para instalar en un sistema embebido, o junto con el kit de desarrollador que incorpora una placa especializada, como puede ser una placa de **Raspberry**, con diferentes entradas y salidas.

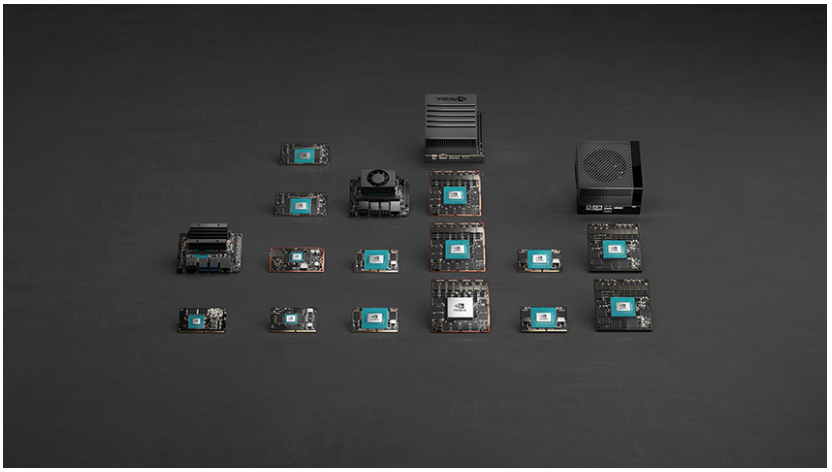


Figura 10: Diferentes dispositivos de la familia **Jetson**

Nvidia pone a disposición una gran variedad de dispositivos para poder elegir lo más conveniente para la tarea que se quiera desarrollar.

- **ASUS** también ha desarrollado una placa para tareas de *machine learning*: **Tinker Edge** (figura 11). Está basada en un producto anterior de **ASUS**, la *Tinker board*, que es un ordenador monoplaca, como la **Raspberry Pi**, que además está diseñada para ser compatible con ésta. La versión **Tinker Edge T** incorpora el acelerador **Edge TPU** de Google visto anteriormente, mientras que la versión **Tinker Edge R** utiliza un acelerador **Rockchip NPU**. Un punto positivo es la posibilidad de utilizar componentes de **Raspberry**, como por ejemplo la cámara, en las placas de **Tinker Edge**.

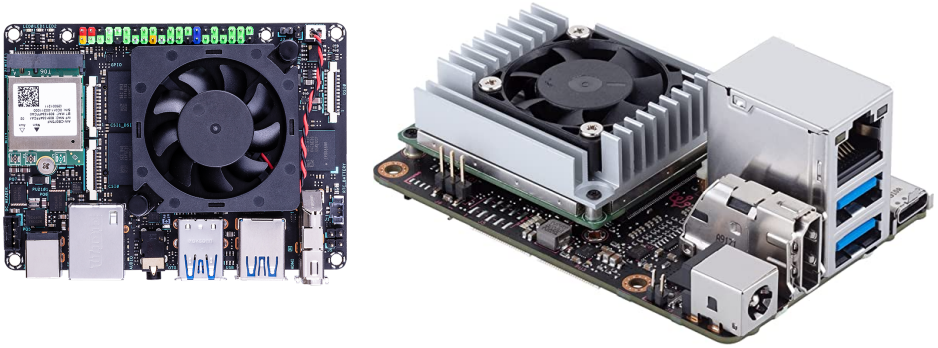


Figura 11: **Tinker Edge R** izquierda, **Tinker Edge T** derecha

- **Intel** ha desarrollado chips específicos para ser utilizados en visión artificial: **Myriad 2** y **Myriad X**. Estos chips son VPUs (*Vision processing unit*), que es un tipo de microprocesador diseñado para acelerar tareas de visión artificial.
  Como producto ofrecen el **Intel Movidius Neural Compute Stick**, que es un USB que tiene la misma funcionalidad que el acelerador USB de **Coral** (figura 12). Éstas, además de soportar el *framework* **TensorFlow**, soporta **Caffe**.

Figura 12: **Movidius Neural Compute Stick** y el chip **Myriad X**

3. Análisis del problema

A continuación se va a proceder a comentar las diferentes soluciones que se plantearon y la solución escogida. También se analizará la seguridad y el plan de trabajo.

3.1 Identificación y análisis de posibles soluciones

Para realizar este TFM, se plantearon varias soluciones:

- Utilizar la **Raspberry Pi** para hacer todos los cálculos y todas las predicciones. Esto significa sobrecargar mucho la placa y no podría funcionar correctamente.

- Ejecutar el modelo en un ordenador y comunicarse con la placa **Raspberry** por medio de la red, pero esto no es algo recomendable, como se explicó en el punto ***2.5 Alternativas***. Que a su vez se tendría un problema de seguridad al estar conectado a la red si se quiere ejecutar tareas sin supervisión del usuario, en el sentido de realizar una tarea de IA durante mucho tiempo donde el usuario no puede estar presente. Además está el problema del consumo, el cual sería elevado al tener un ordenador encendido todas las horas del día.

- Utilizar una TPU de **Intel** o **Nvidia** y ejecutar los modelos en las mismas. Esto permite utilizar las TPUs para todos los cálculos y la Raspberry Pi no sería de mucha utilidad.

- Utilizar el acelerador USB de Coral y comunicar el resultado de la predicción a la **Raspberry Pi**.

3.2 Solución propuesta

Vistas las alternativas disponibles y los problemas que pueden surgir, se pretende encontrar una forma de realizar *Deep Learning* con un *hardware* limitado, que además sea portable, asequible y que tenga un consumo bajo.
Con estos requisitos, se decidió elegir el acelerador USB **Coral** por varios motivos:

I. Por ser más asequible que las demás opciones, con un valor de 75€ iniciales, actualmente se encuentra a 50€, mientras que las demás opciones superan los 100€.

II. Desde el año pasado existe una escasez de microcontroladores, lo que resulta en encontrar dispositivos como estos, una tarea difícil[34]. El acelerador USB **Coral**

era de los pocos que era posible adquirir. Por suerte, este problema parece que empieza a mejorar y es posible de nuevo adquirir los demás dispositivos si se desea[35].

III. La posibilidad de poder tener un sistema con inteligencia artificial en un microordenador sin necesidad de *hardware* específico, a parte del mismo acelerador, que se pueda incorporar a sistemas ya existentes. De esta forma, teniendo los módulos de **Python** correctamente instalados, se pueden añadir nuevas tareas de inteligencia artificial a sistemas que están siendo utilizados para otras tareas. Siendo esto uno de los objetivos del TFM, ya que si se utiliza una placa de **Nvidia** o **ASUS**, el sistema ya está montado y se tendría que rehacer, instalar y configurar las funciones y tareas existentes en el sistema actual.

IV. Al ser productos relativamente nuevos, la comunidad en torno a estos dispositivos es inferior a **Raspberry**, sobre todo en caso de errores, donde es más fácil encontrar soluciones con comunidades mayores.

V. Actualmente no se puede conectar una **Raspberry Pi** con **Jetson**, por lo que para realizar transferencias de datos es necesario utilizar un cable **Ethernet** o **Wi-Fi**.

VI. Un punto negativo de esta solución es que los modelos utilizados con **Coral** deben ser cuantizados, habiendo modelos que utilizan operaciones que no son soportadas por esta técnica, como la función de activación *Leaky ReLu*(23). Por tanto, se tiene un menor número de modelos compatibles. Pero a pesar de esto, siguen habiendo modelos con suficiente precisión en la predicción para ser una solución acertada.

3.3 Análisis de seguridad

En cuanto a seguridad, como se ha sugerido a lo largo de los puntos anteriores, puede existir un problema de seguridad al utilizar la red, donde los atacantes pueden acceder al dispositivo o incluso a otros dispositivos y ordenadores que comparten la misma red. En el caso de la solución propuesta, el dispositivo con el acelerador USB no tiene necesidad de estar conectado a la red para ejecutar operaciones y guardar información. Es posible tener acceso a la red para realizar ciertas acciones, pero no es un requerimiento.

Por otra parte, al ser un microordenador, se dispone de más funciones de seguridad, a diferencia de un dispositivo IoT que se diseña para ser lo más eficiente posible y de reducido tamaño, por lo que sus funciones de seguridad son limitadas.

Si no se activa la red, *bluetooth*[24] u otro método de conexión, la única forma de acceder al dispositivo es de forma física. En el punto ***5º Desarrollo de la solución*** se explicará con más detalle la forma de conectarse, pero es posible conectarse solamente con un cable HDMI para configurar, cambiar o mirar la placa **Raspberry** y las tareas que debe de realizar. Esto puede ser un punto negativo, pero como todos los ordenadores, si se tiene acceso físico al mismo, ya no hay seguridad que valga.

3.4 Plan de trabajo

En cuanto al plan de trabajo, inicialmente se definió de la siguiente forma:

Tarea a realizar	Tiempo	Razonamiento
Leer el artículo[36] y destacar las partes clave	2 días	Es un artículo largo y muchos puntos a destacar
Realizar las instalaciones del sistema operativo en la placa **Raspberry** y de *software* necesarias	4 días	Hay que instalar varias cosas además del sistema operativo, y siempre suele haber problemas de algún tipo, por lo que se da un poco más de márgen
Montar el coche **PiCar** y comprobar su funcionamiento	2 horas	Según el mismo artículo, para montar el coche se tardan un par de horas, pero al tener el coche montado, solo faltaba conectar los cables y probar su funcionamiento
Desarrollar los *scripts*[25] de **Python**, hacer el circuito y realizar la primera parte donde el coche recorre el circuito detectando las líneas	2 días	Los *scripts* están disponibles en el repositorio de **Github**[26] y solo habría que descargarlos y configurar en la placa
Obtener los datos necesarios y realizar la segunda parte donde se enseña al coche **PiCar** a recorrer el circuito	3 días	Con los explicado en el artículo, no parece haber mucha dificultad
Obtener los datos necesarios y realizar la tercera parte donde se enseña al coche **PiCar** a reconocer las señales de tráfico	5 días	Obtener las imágenes de las señales y etiquetarlas, no es mucho trabajo, pero el proceso puede complicarse un poco y abarcar un tiempo mayor
Obtener los datos de la carretera y señales con un coche real	2 días	Requiere desplazarse en coche y obtener suficiente imágenes buenas
Preparar los datos, entrenar los diferentes modelos y ejecutarlos en un programa que se debe diseñar y escribir	7 días	Al igual que en la tercera parte, pero además se tiene más imágenes y se tiene que hacer un pequeño programa para mostrar los resultados

Sugerir modificaciones y optimizaciones	Tiempo sobrante	Se plantearía hacer otros modelos e intentar mejorar los resultados obtenidos

Este es el plan de trabajo inicialmente pensado, pero desde el primer momento surgieron numerosos problemas, todos ellos explicados en el punto ***5º Desarrollo de la solución***.
La mayor parte del tiempo se gastó en la configuración e instalación inicial. En la tarea que tiene un tiempo estimado de cuatro días, el tiempo real fueron de unos dos meses. Al principio se apuntó todos los comandos, cambios, instalaciones ejecutadas y tiempos invertidos. Pero, al no encontrar una solución, se empezó a trabajar sin tener nada de esto en cuenta, sabiendo que cada comando y cambio que se realice, puede que no afecte en nada al resultado final. También se rehizo la parte de la instalación y configuración de la placa **Raspberry** varias veces.
Una vez el problema de la configuración e instalación estaba resuelto, surgían problemas con el código, desde el utilizado por el coche **PiCar** como en **Colab**. Al igual que antes, se realizaron muchos cambios y se probó todo tipo de soluciones para que funcionara correctamente, por lo que tampoco se apuntaron los tiempos, por lo que cada paso se convirtieron en días o semanas de búsquedas y pruebas. Como se comentó anteriormente en el punto ***2º Estado del arte***, la inteligencia artificial está teniendo muchos avances y, como consecuencia, los dispositivos *hardware* y el *software* están en constante evolución. Existen muchos artículos y guías escritos hace unos pocos años, que ahora mismo son más una idea de lo qué se puede hacer y una sugerencia de qué utilizar, más que una guía a seguir.
Destacar también que en **Colab**, la versión 1 de **TensorFlow** ya no está soportada desde este Agosto, y no se puede utilizar el comando `%tensorflow_version 1.x` como se hacía siempre. Además, por lo visto, se han realizado otros cambios que han afectado a los módulos usados durante el desarrollo de este TFM. La tercera parte, donde se realiza el modelo para las señales, utiliza un módulo que actualmente en **Colab** tarda horas en descargarse e instalarse, hasta el punto que la sesión se desconecta por superar el tiempo permitido de uso. La solución a este problema se encontró a finales del mes de Agosto y está cambiado en el código de **Colab** final.

4. Diseño de la solución

En este apartado se describirán los diferentes elementos, tanto *hardware* como *software*, utilizados en la realización del TFM. También se describirá brevemente su interconexión y funcionamiento y las fases de desarrollo.

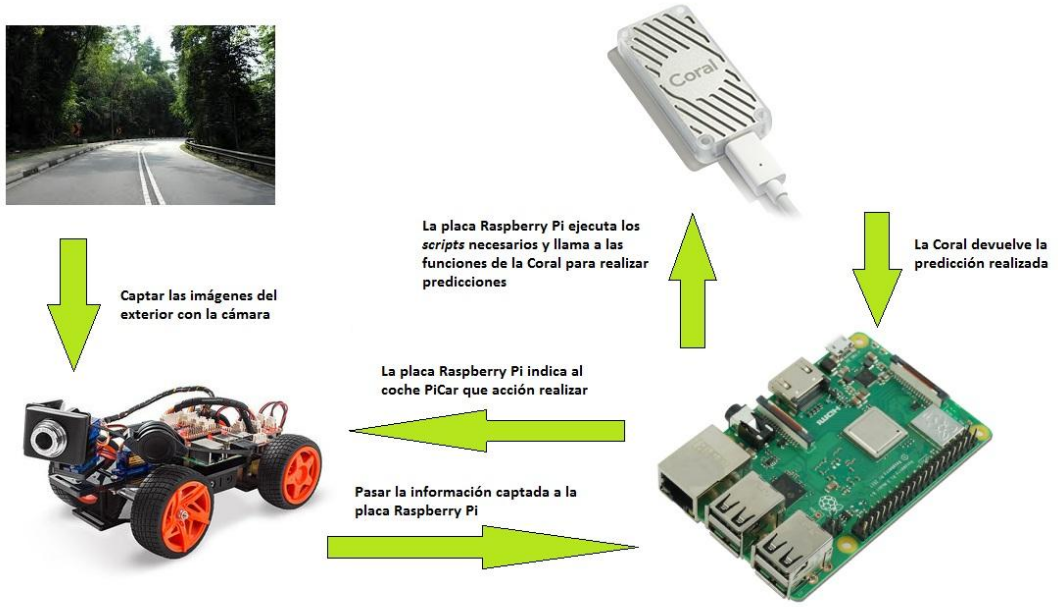


Figura 13: Funcionamiento de la solución

Se entrena un modelo en **Colab** con imágenes obtenidas de la cámara del coche **PiCar** sobre el circuito, y etiquetando la imágen con el valor de giro de las ruedas frontales para que se pueda relacionar la situación del giro con un número para ser utilizado posteriormente. Para que reconozca las señales de tráfico, se realizan pasos similares pero se crearán ficheros donde se encuentren las etiquetas de los diferentes objetos que componen la imágen, que son coordenadas en X y en Y del comienzo y final de un rectángulo que contiene el objeto.

Con los modelos entrenados en **Colab**, se transfieren a la placa **Raspberry Pi**, la cual estará montada sobre el coche **PiCar**, y tendrá conectada una cámara. Ésta se encargará de obtener las imágenes de la parte frontal del coche **PiCar** para ser procesadas posteriormente y obtener una predicción del modelo, momento en el cual, el coche **PiCar** ejecutará una acción conveniente: girar, parar o acelerar o desacelerar. El acelerador USB de **Coral AI** será el encargado de realizar los cálculos del modelo y obtener la predicción. Este proceso se puede observar en la figura 13.

El coche real tendrá un funcionamiento similar. Ahora bien, como no es posible interconectar la placa al coche en sí para que esta afecte al funcionamiento del mismo, se creará un programa sencillo que, obteniendo los valores de las predicciones, indique

por pantalla qué acción se debe realizar a continuación: girar, parar o acelerar o desacelerar, como con el coche **PiCar**.

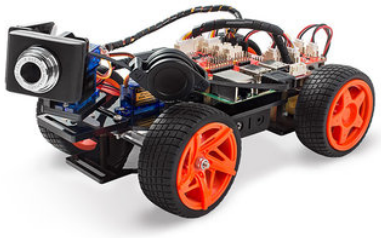
4.1 Hardware

4.1.1 Cinta adhesiva (figura 14)

La cinta adhesiva se utilizará para crear un circuito el cual el coche **PiCar** recorrerá. Se puede utilizar otro material u objeto, pero tiene que tener una diferencia de color significativa. Se explicará con más detalle en el punto **5º Desarrollo de la solución**.

Figura 14: Cinta adhesiva

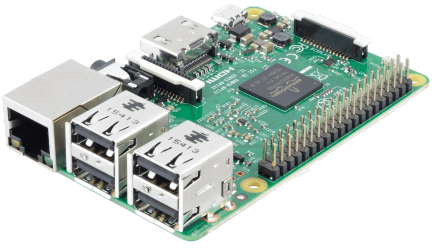
4.1.2 Coche PiCar (figura 15)



Es el coche que recorrerá el circuito y reconocerá señales utilizando el modelo entrenado. El coche **PiCar** es un robot de código abierto basado en **Raspberry Pi**.

Figura 15: Coche PiCar

4.1.3 Raspberry Pi (figura 16)



Una placa **Raspberry Pi** que irá montada sobre el coche **PiCar** y tendrá conectada la cámara y el acelerador USB de **Coral AI**. La placa funcionará con un sistema operativo Linux y controlará los diferentes componentes y ejecutará los *scripts* necesarios.

Figura 16: Raspberry Pi

4.1.4 Fuente de alimentación (figura 17)

La fuente de alimentación de 5.1 Voltios y 2,5 Amperios inicialmente se utilizará para configurar la **Raspberry Pi** y programar el funcionamiento, además de realizar pruebas. Posteriormente se utilizarán dos baterías para tener el resultado final.

Figura 17: Fuente de alimentación

4.1.5 Baterías (figura 18)

Baterías recargables de 3.7 Voltios y 2.6 Amperios que se utilizarán para dar autonomía al coche **PiCar**.

Figura 18: Baterías

4.1.6 Cámara (figura 19)

Una cámara incluida con el coche **PiCar** que se utilizará para capturar imágenes de la dirección del coche y que serán enviadas a la **Raspberry Pi** para su procesamiento.

Figura 19: Cámara

4.1.7 Cable HDMI (figura 20)

El cable HDMI permitirá conectar la **Raspberry Pi** a una pantalla para poder realizar la configuración inicial y activar el **VNC** (*Virtual Network Computing*) para poder conectarnos a la placa remotamente sin necesidad de que esta esté conectada a una pantalla.

Figura 20: Cable HDMI

4.1.8 Tarjeta SD (figura 21)

Tarjeta de memoria *Secure Digital* para dispositivos portátiles como teléfonos móviles, microordenadores o ordenadores portátiles.

Figura 21: Tarjeta SD

4.1.9 *USB Accelerator* de Coral AI (figura 22)

Lo que se utilizará será el *USB Accelerator* que distribuye **Coral**, ya que también ofrece otros productos con diferentes nombres. Pero, para diferenciar el acelerador USB de los demás componentes que puede resultar confuso, y mayor brevedad a la hora de leer, se utilizará **Coral AI** o **Coral** para referenciar este componente.

Figura 22: Acelerador USB

La **Coral** se utilizará para hacer los cálculos necesarios y obtener una predicción cuando se utilicen los modelos entrenados. Es capaz de realizar cuatro trillones de operaciones por segundo utilizando 0,5W por trillón de operaciones, es compatible con sistemas operativos **Windows**, **Mac** y **Linux** y soporta **TensorFlow**.

4.2 Software

4.2.1 Sistema operativo

Se utilizará un ordenador con sistema operativo basado en **Linux** para conectarse a la **Raspberry Pi** y programar los diferentes *scripts*. Es posible realizar los mismos pasos de este TFM con cualquier otro sistema operativo.

4.2.2 Python (figura 23)

Figura 23: Python

Python es el lenguaje de programación utilizado para programar tanto el comportamiento del coche **PiCar** como el código utilizado en **Colab** donde se entrenará los modelos. **Python** es uno de los lenguajes más utilizados en el campo de la visión artificial con módulos como **OpenCV** y **TensorFlow** siendo estos de los más destacados y que se utilizarán en este TFM.

4.2.3 Colab (figura 24)

Figura 24: Colab

También llamado *Colaboratory*, es una herramienta de **Google** que permite escribir código en **Python** en celdas y ejecutarlas obteniendo el resultado. Una similitud sería tener el intérprete de **Python** que se tiene en la terminal, pero con mayor libertad y la posibilidad de añadir bloques de texto.

Además, nos permite conectarnos a una GPU y TPU de forma gratuita sin necesidad de configurar nada adicionalmente a lo requerido por el programa que utilicemos, por ejemplo, no hay necesidad de instalar y configurar cuDNN[27] que puede ser algo complicado de hacer si queremos utilizar una GPU propia.

4.2.4 RealVNC (figura 25)

Figura: 25: RealVNC

Herramienta *software* utilizada para conectarse con la **Raspberry Pi** de forma remota sin necesidad de disponer de pantalla.

4.2.5 LabelImg (figura 26)

Herramienta *software* para definir las etiquetas de las imágenes. En el punto **5.5** se explicará con más detalle su uso.

Figura 26: LabelImg

5. Desarrollo de la solución

En este punto se procederá a explicar el proceso seguido para la configuración, instalación y desarrollo de código seguidos para obtener el resultado final: un microordenador realizando tareas de inteligencia artificial (*Deep Learning*) de forma autónoma y con precisión.

5.1 Configuración Raspberry Pi

Para empezar, hay que preparar el sistema operativo de **Raspberry Pi**, siguiendo los pasos indicados en la web de **Raspberry**[37]. Es necesaria una placa **Raspberry Pi**, **3 B** en este caso, una tarjeta *SD*, un cable HDMI, un ratón, teclado, y una fuente de alimentación. En la guía oficial, mencionada anteriormente, aparecen dos imágenes de fuentes de alimentación, en la primera aparece un cargador que acepta entre 100 y 240 Voltios y 0,3 Amperios a 50/60 Herzios, y con salida de 5.1 Voltios y 2,5 Amperios, y la segunda imágen una fuente de alimentación similar pero con menor amperaje en la entrada y 5 Voltios en la salida. Al principio de la realización de este TFM, se utilizó una fuente casi identifica a la segunda imágen, la única diferencia se encuentra en el amperaje de salida, que se tenía 2 Amperios. Con esta fuente de alimentación se tenían serios problemas, a veces la **Raspberry Pi** no podía iniciarse con todos los componentes conectados, siendo el coche **PiCar** y la **Coral AI**, otras veces la tarjeta *SD* se corrompía y se perdían todos los datos incluido el sistema operativo. Tras buscar posibles soluciones al problema del *boot*(28) de la **Raspberry Pi** y de la corrupción de memoria, en el foro oficial de **Raspberry Pi** existe un *post*(29) con los posibles errores que se pueden producir y cómo solucionarlos[38]. La gran mayoría de problemas surgen por una mala fuente de alimentación, por lo que se decidió cambiar la fuente de alimentación actual a la oficial recomendada por **Raspberry Pi** que aparece en la primera imágen de la guía y, efectivamente, no se ha tenido ningún problema *hardware* con la **Raspberry Pi** desde entonces.

Se descarga el **Imager** de **Raspberry Pi** desde su página web oficial[39]. Una vez descargado e instalado, se inicia el programa y aparecen dos opciones para seleccionar un sistema operativo y un lugar donde instalar el sistema operativo. Al seleccionar la opción del sistema operativo, aparecerán varias opciones, siendo la primera la más intuitiva de seleccionar e instalar (figura 27).

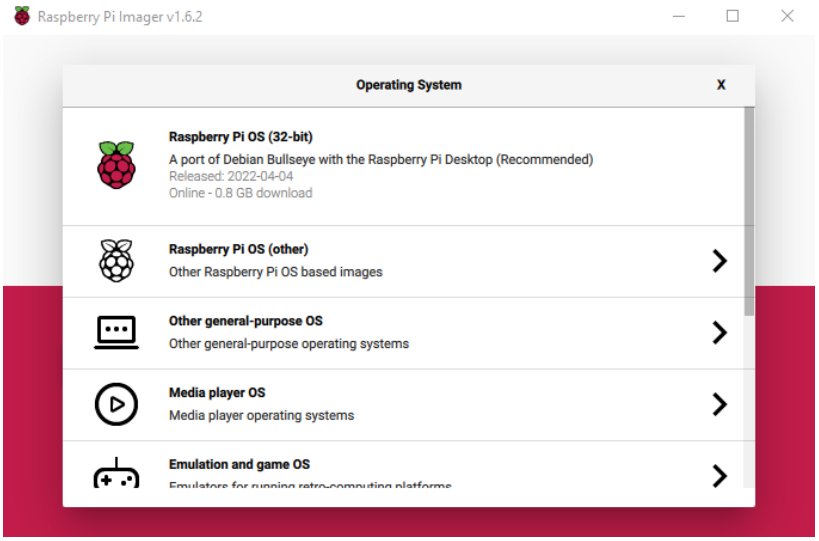


Figura 27: Menú del **Imager**

Esta opción no es recomendable para este TFM, ya que esa versión tiene incluido **Python 3.9** por defecto y, por el momento, no está soportado por **Coral AI**, por lo que es necesario una versión anterior de **Python**. Más adelante se comentarán los pasos seguidos para intentar hacer funcionar **Coral AI** con el primer sistema operativo mostrado en la figura 28 y **Python 3.9**, sin éxito.

Todos los sistemas operativos disponibles en la segunda opción tienen **Python 3.9** por defecto, por lo que es necesario conseguir un sistema operativo anterior con la versión 3.8 de **Python**. Para esto hay que acceder a su repositorio de imágenes[40] y descargar un fichero *.zip* de una versión que tenga **Python 3.8** por defecto. En este caso se eligió la versión con fecha 2020-02-13. Se extrae el fichero con extensión *.img* y en el **Imager** se selecciona la última opción "*Use custom*" (figura 28).

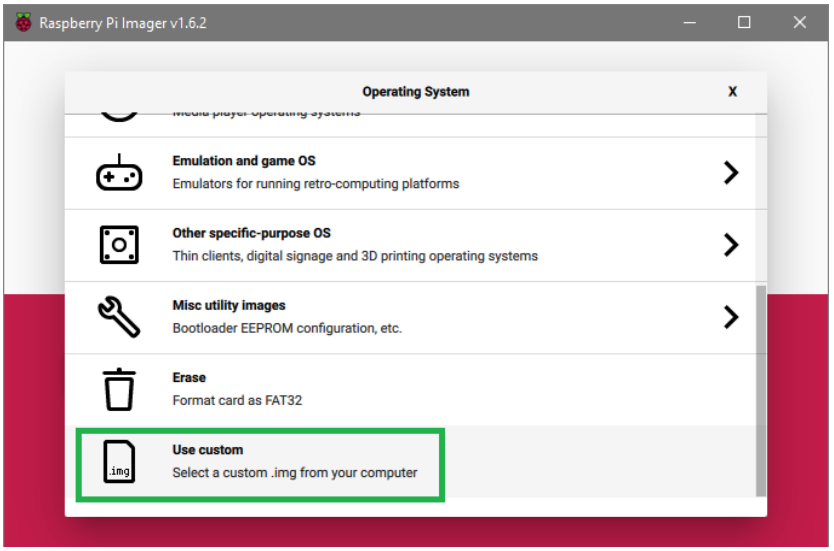


Figura 28: Opción *"Use custom"*

Seleccionar el fichero *.img* y a continuación la tarjeta *SD* que se haya introducido para instalar el sistema operativo.

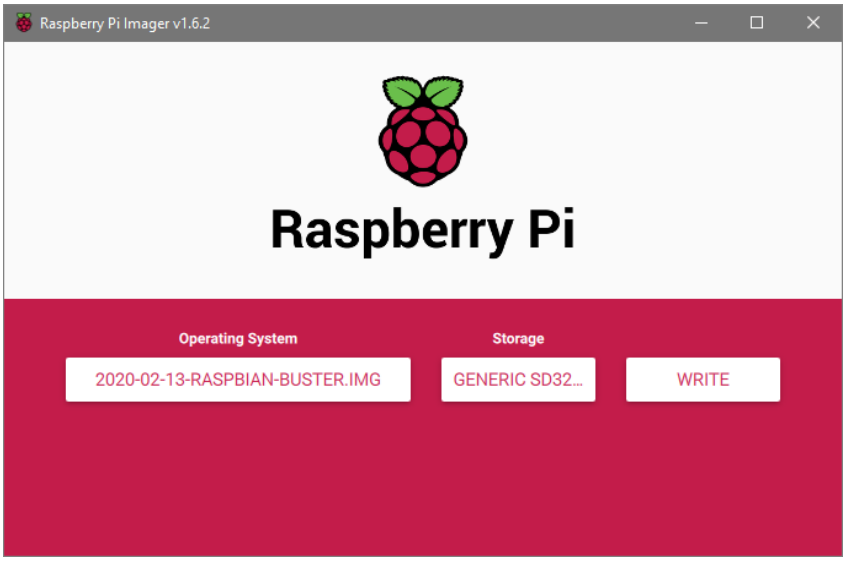


Figura 29: Vista resultante antes de generar el sistema operativo en la tarjeta *SD*

Seleccionar *"Write"* (figura 29) y esperar a que finalice el proceso, momento en el cual, se puede retirar la tarjeta *SD* (figura 30).

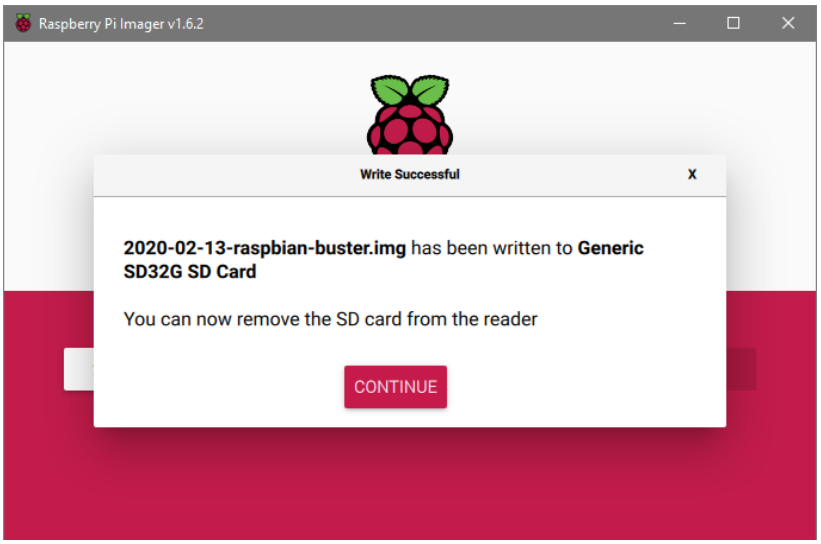


Figura 30: Resultado final indicando la extracción de la tarjeta *SD*

Una vez el proceso ha terminado, se saca la tarjeta *SD* de forma segura y se introduce en la **Raspberry Pi**, se conecta el cable HDMI a la misma y a una pantalla, se conecta un ratón y teclado y a continuación la fuente de alimentación, encendiendo la **Raspberry Pi**. Se deja que termine el proceso de *boot* y debería presentarse la pantalla inicial del sistema operativo.

Existe la probabilidad de que durante este paso no se realice el *boot* o que no aparezca la pantalla inicial apareciendo un error por pantalla. Si es el caso, se debe formatear la tarjeta *SD* y volver a realizar los pasos para instalar el sistema operativo. Esto ocurre solamente la primera vez que se instala el sistema operativo, una vez se llega a la pantalla de inicio, el problema no aparece más.

Ahora se tienen que realizar los pasos indicados e instalar la actualización que aparece para el sistema operativo instalado. Es probable que no se descargue ni instale nada al ser un sistema operativo antiguo. Al terminar, se reinicia la **Raspberry** y se puede configurar la misma para acceder por medio de **VNCviewer**, previamente instalado en nuestro ordenador, **Linux** en este caso. Para instalar el programa, se debe acceder a la página de **RealVNC**[41], elegir el sistema operativo donde se quiera instalar y tras finalizar la descarga, se instala.

Lo primero es abrir una terminal y acceder a la configuración de **Raspberry** con el comando `sudo raspi-config`, nos dirigimos a la opción ***3.Interface options*** y activamos *SSH*(30) y *VNC*. Ahora se mira la dirección ip de la **Raspberry** con el comando `ifconfig` y con esta dirección ip se puede conectar remotamente a la placa con el programa **VNCviewer**.
Con el programa abierto, se elige la opción *"File"* y *"New connection…"* (figura 31).

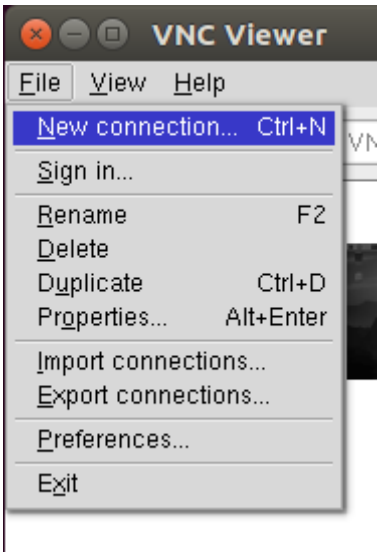


Figura 31: Crear una nueva conexión

Se abrirá una ventana donde hay que introducir la dirección ip de la placa y un nombre que se desee (figura 32).

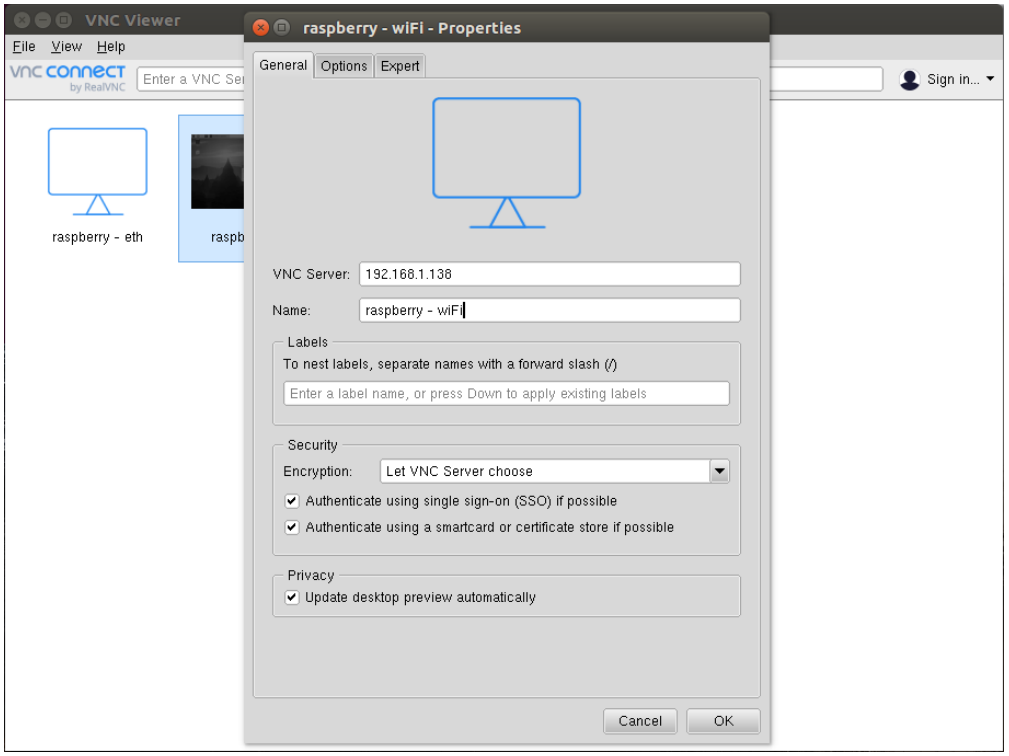


Figura 32: Ventana con las propiedades de la conexión

Se acepta y se prueba conectarse. Aparecerá una ventana de autenticación donde se deberá introducir el nombre de la placa, "***pi***" en este caso, y la contraseña que se haya introducido cuando se configuró la **Raspberry** inicialmente (figura 33).

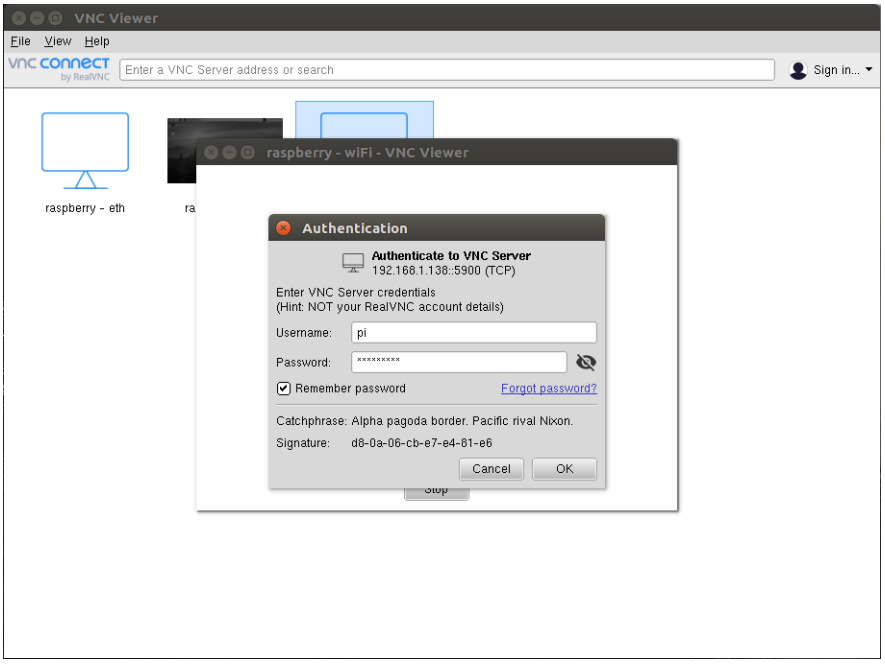


Figura 33: Ventana de autenticación

Si todo está correcto, aparecerá el escritorio de la placa, momento en el que se puede quitar la pantalla, el ratón y el teclado dejando solo la **Raspberry Pi** y el Wi-Fi encendido.
Cabe destacar que al apagar y volver a iniciar la placa sin conectarla a ninguna pantalla, la resolución cambiará a una muy baja. Para cambiar esto, se debe ejecutar el comando `sudo raspi-config` de nuevo y seleccionar la opción ***7.Advanced Options*** como se indica en la figura 34.

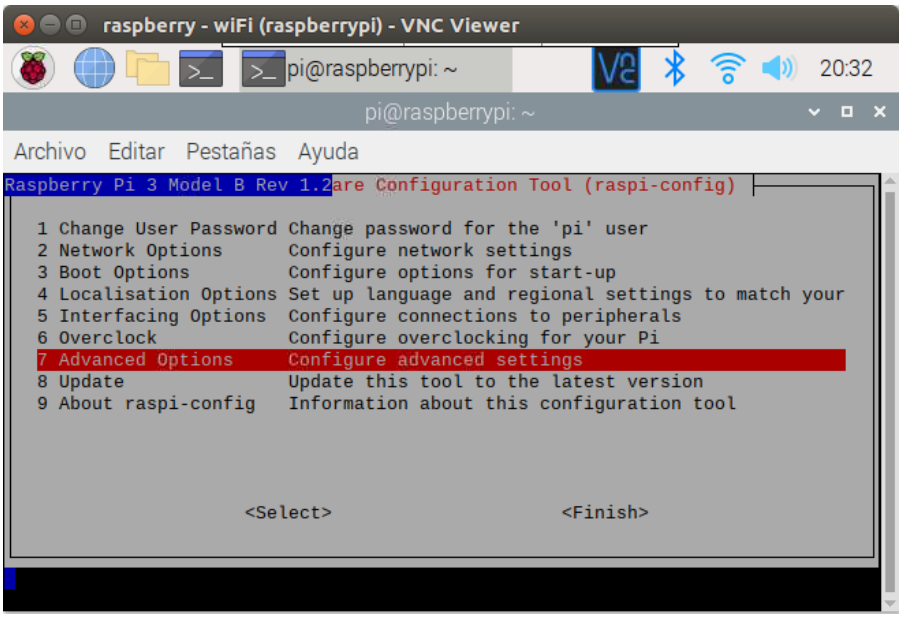


Figura 34: Opción ***7.Advanced Options***

Se selecciona la opción *"**A5 Resolution**"* (figura 35).

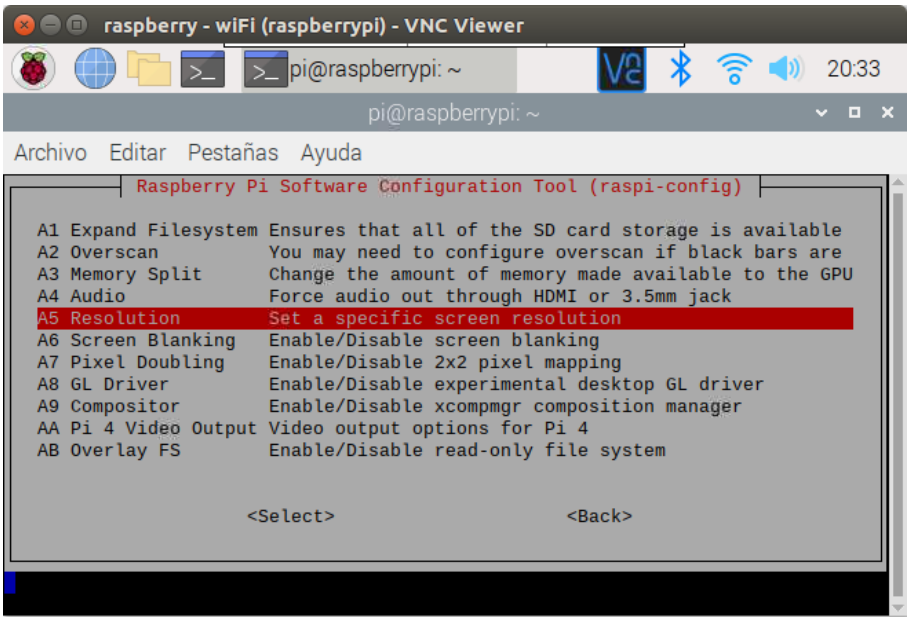


Figura 35: Opción ***A5 Resolution***

Y se configura la resolución que se desee (figura 36).

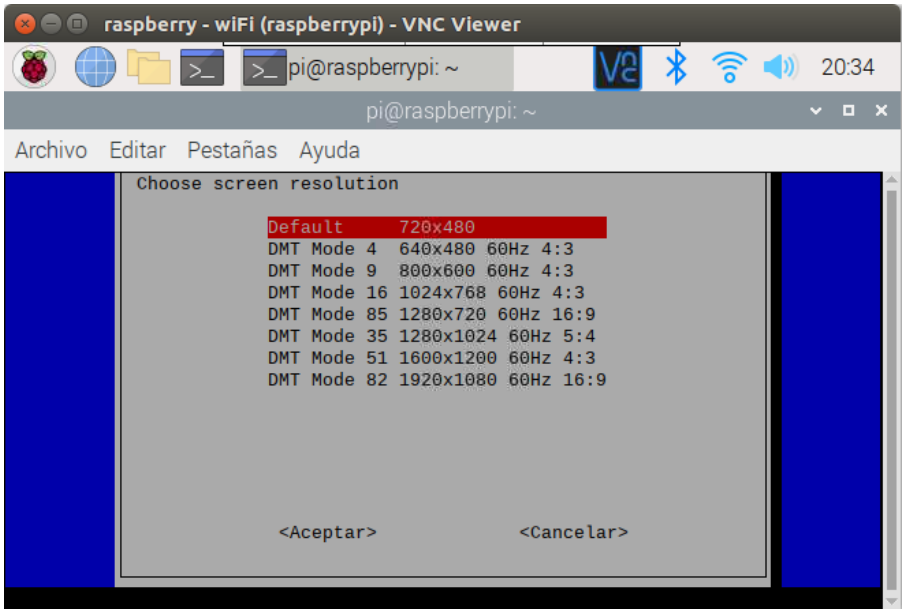


Figura 36: Diferentes resoluciones

5.2 Instalación de *software* y pruebas del coche PiCar

En el artículo inicial, se muestra como se configura un sistema de ficheros remoto para acceder a ficheros desde el ordenador y la **Raspberry Pi**. En este caso, se va a ignorar esta parte ya que se utilizará *SSH* para la transferencia de ficheros, pero también es posible configurar el sistema de ficheros remoto si se requiere.

Ahora se realiza un clonado del repositorio que se indica:

```
git clone https://github.com/dctian/SunFounder_PiCar.git
```

Al finalizar, se accede al directorio **~/SunFounder_PiCar/picar/** y se realiza otro clonado de repositorio (figura 37):

```
git clone https://github.com/dctian/SunFounder_PCA9685.git
```

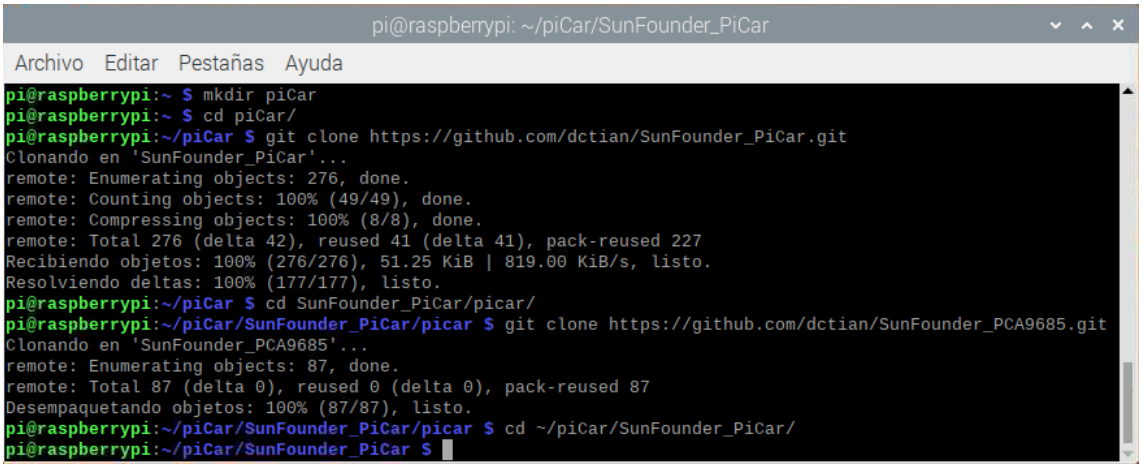


Figura 37: Clonado de repositorios

Antes de continuar, se tiene que ejecutar el comando `sudo apt-get update` para actualizar el sistema operativo (figura 38). Al ser una versión antigua del sistema operativo, aparecerá el mensaje que se muestra en la figura 38.

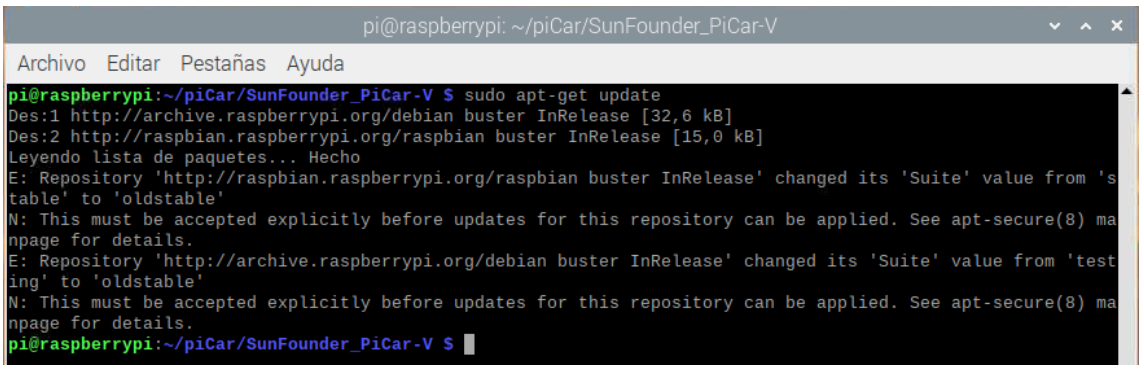


Figura 38: Mensaje indicando que se tiene un versión antigua

Esto se soluciona con el siguiente comando (figura 39):

```
sudo apt update -allow-releaseinfo-change
```

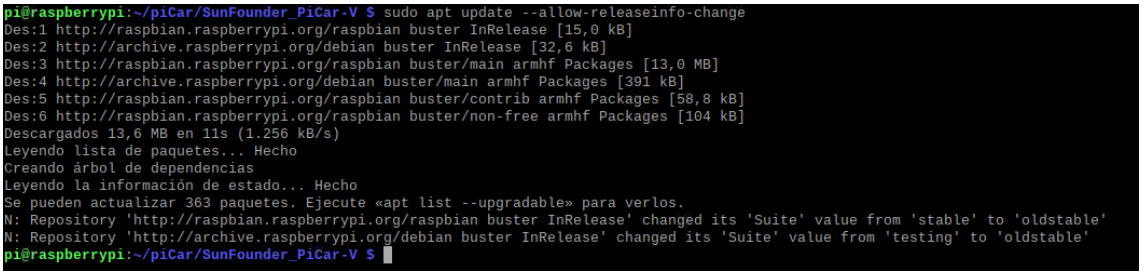


Figura 39: Solución al problema

Ahora es posible actualizar, tardando bastante tiempo en finalizar, y continuar con la instalación. Se ejecuta el comando `sudo python setup.py install` mostrado y, en el directorio principal, accediendo con el comando `cd`, se ejecuta otro clonado con

```
git clone https://github.com/dctian/SunFounder_PiCar-V.git
```

se accede al directorio /SunFounder_PiCar-V/ y se ejecuta el comando

```
sudo ./install_dependencies
```

Al terminar, aparecerán los módulos **django**, **python-smbus**, **python-opencv** y **libjpeg8-dev** instalados correctamente, se escribe *"yes"* para reiniciar (figura 40).

```
Installed /usr/local/lib/python2.7/dist-packages/SunFounder_PiCar-1.0.1-py2.7.egg
Processing dependencies for SunFounder-PiCar==1.0.1
Finished processing dependencies for SunFounder-PiCar==1.0.1
complete

Copy MJPG-Streamer to an Alternate Location.

complete

Enalbe I2C

    Seem i2c_arm parameter already set, skip this step

complete

Installation result:
django   Success
python-smbus  Success
python-opencv  Success
libjpeg8-dev  Success
The stuff you have change may need reboot to take effect.
Do you want to reboot immediately? (yes/no)
```

Figura 40: Módulos instalados con el mensaje para reiniciar la placa

La duración del montaje del coche **PiCar** se estima en dos horas, pero en mi caso recibí el coche **PiCar** ya montado, a falta de conectar los cables del coche a la placa **Raspberry**. Para conectar los cables, se utilizó un manual puesto a disposición por la misma compañía que ha desarrollado el coche **PiCar**[42].

Con los cables conectados a la placa, ya se puede probar el funcionamiento del coche. Para esto se abre una terminal y se accede al entorno de **Python** con el comando `python3`. Es importante indicar que sea la versión 3 para que no se ejecute la versión 2.7 que también se encuentra instalada por defecto con el sistema operativo. Se importa el módulo `picar` y se inicializa el coche con `picar.setup()`.

Para probar el funcionamiento de las ruedas frontales, se ejecuta `picar.front_wheels.test()`, donde las ruedas deberían girar a la izquierda, colocarse en el centro, a la derecha y repetir los movimientos a la inversa.

Para probar las ruedas traseras, se ejecuta el comando `picar.back_wheels.test()`, en este caso el coche acelerará hacia delante y luego hacía atrás.

Si las ruedas se mueven, se puede confirmar que el módulo del coche **PiCar** está correctamente instalado.

En la siguiente parte se comienza instalando una serie de bibliotecas relacionadas con **OpenCV**, siendo éste un módulo de **Python** para visión artificial capaz de modificar y transformar imágenes y vídeos. Se instalan uno a uno para evitar problemas:

```
sudo apt-get install libhdf5-dev
sudo apt-get install libhdf5-serial-dev -y
sudo apt-get install libatlas-base-dev -y
sudo apt-get install libatlas-base-dev
sudo apt-get install libjasper-dev -y
sudo apt-get install libqtgui4 -y
sudo apt-get install libqt4-test -y
```

Y luego se instala **OpenCV** y **matplotlib**, que es un módulo para hacer gráficos entre otras funciones. Para ello se utilizan los comandos `pip3 install opencv-python` y `pip3 install matplotlib`. Es probable que estos módulos ya estén instalados de pasos anteriores, específicamente **OpenCV** que se instaló con el comando `sudo ./install_dependencies`.

Se hace un clonado del repositorio donde se encuentran todos los *scripts* relacionados con el funcionamiento del coche **PiCar** (figura 41) y se realiza una prueba con la cámara para comprobar que todos los módulos están correctamente instalados. El resultado tiene que ser como el de la figura 42.

```
pi@raspberrypi:~/piCar $ git clone https://github.com/dctian/DeepPiCar.git
Clonando en 'DeepPiCar'...
remote: Enumerating objects: 1027, done.
remote: Total 1027 (delta 0), reused 0 (delta 0), pack-reused 1027
Recibiendo objetos: 100% (1027/1027), 141.14 MiB | 3.57 MiB/s, listo.
Resolviendo deltas: 100% (320/320), listo.
Revisando archivos: 100% (546/546), listo.
pi@raspberrypi:~/piCar $ ls
DeepPiCar  SunFounder_PiCar  SunFounder_PiCar-V
pi@raspberrypi:~/piCar $
```

Figura 41: Clonado de repositorio

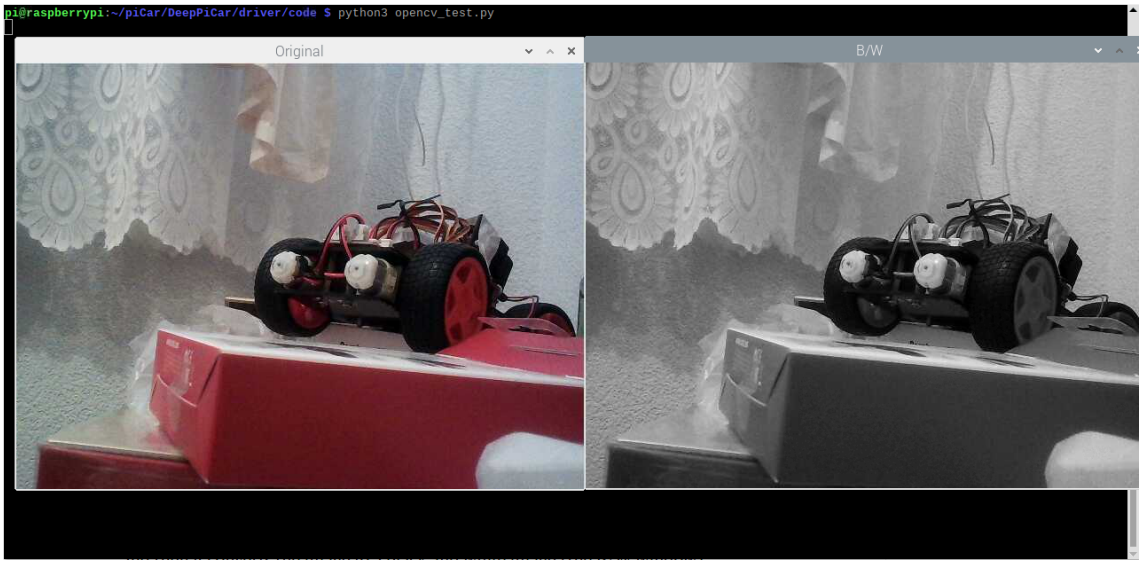


Figura 42: Resultado de la prueba

Una vez esté todo correcto, se puede proceder a instalar las bibliotecas para hacer funcionar la **Coral AI**. Destacar que los módulos de **TensorFlow** y **Keras** ya no están separados, éste último se incluye junto a **TensorFlow**, y las llamadas a la *API*[31] de **Coral AI** están obsoletas y se requiere utilizar **PyCoral** para hacer dichas llamadas. Esto se utilizará más adelante, pero para realizar la prueba de funcionamiento se utilizará el módulo de **edgetpu**.
Por tanto, solamente es necesario instalar **TensorFlow** con el comando pip, el módulo de **Keras** estará incluido.
Si se sigue como se indica, tras reiniciar e intentar realizar la prueba, aparecerá un error de que el módulo de **edgetpu** no se puede encontrar.
Para instalar correctamente el módulo y sus dependencias, se deben ejecutar los siguientes comandos:

```
echo "deb https://packages.cloud.google.com/apt \
coral-edgetpu-stable main" | sudo tee \
/etc/apt/sources.list.d/coral-edgetpu.list
```

```
curl https://packages.cloud.google.com/apt/doc/apt-key.gpg
| \ sudo apt-key add -
```

```
sudo apt-get update
```

```
sudo apt-get install libedgetpu1-std
```

```
sudo apt-get install python3-pycoral
```

```
mkdir coral && cd coral
```

```
git clone https://github.com/google-coral/pycoral.git
```

```
cd pycoral/
```

```
bash examples/install_requirements.sh classify_image.py
```

En la figura 43, se pueden ver la salida de algunos de ellos.

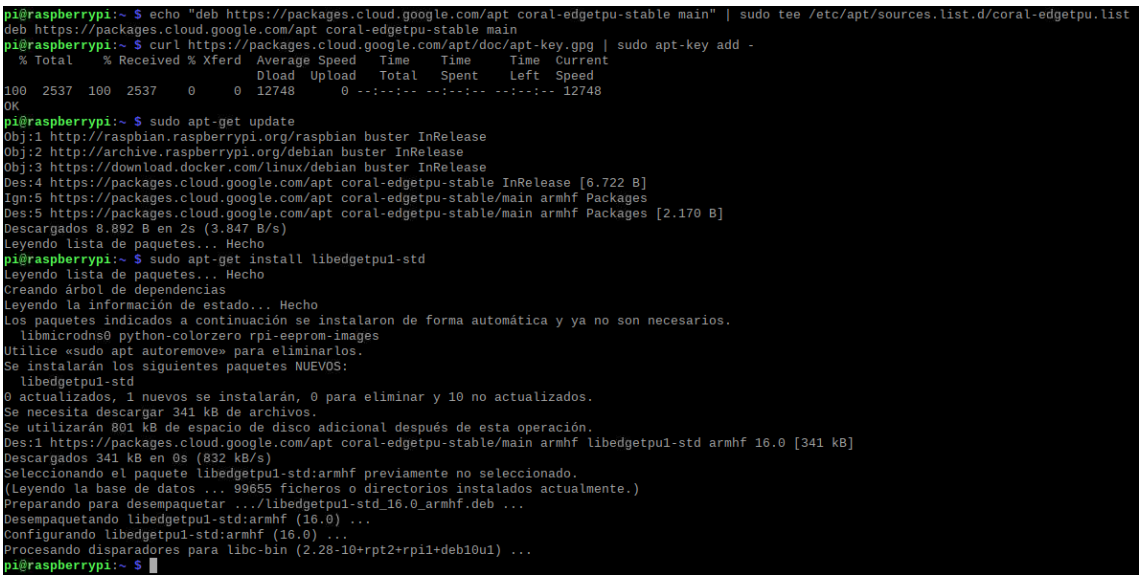


Figura 43: Algunos comando mostrando su salida

Y se prueba que funciona correctamente ejecutando el comando (figura 44):

```
python3 examples/classify_image.py --model \
test_data/mobilenet_v2_1.0_224_inat_bird_quant_edgetpu.tfli
te \ --labels test_data/inat_bird_labels.txt --input \
test_data/parrot.jpg
```

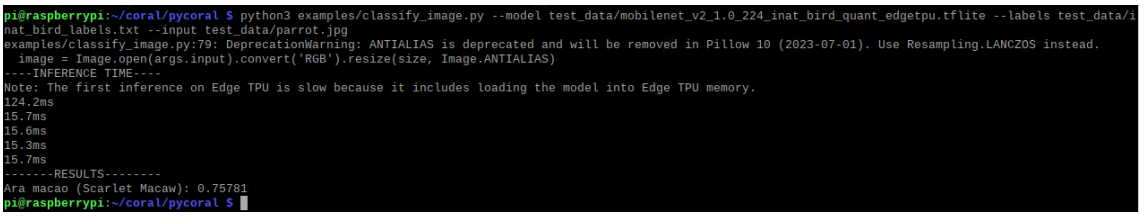


Figura 44: Salida del comando de la prueba realizada

Después de realizar todos estos pasos, falta instalar el módulo de **edgetpu** con el comando:

```
sudo apt-get install python3-edgetpu
```

Realizado esto, se puede comprobar el funcionamiento de la **Coral AI** ejecutando el *script* de prueba en el directorio

```
~/piCar/DeepPiCar/models/object_detection/code/
coco_object_detection.py
```

Pero antes, se debe cambiar la ruta del modelo utilizado por el *script* abriéndolo con un editor de texto como por ejemplo **nano** (figura 45).

```
  GNU nano 3.2                                          code/coco_object_detection.p

"""A demo to classify Raspberry Pi camera stream."""
import argparse
import time

import numpy as np
import os
import datetime

import edgetpu.detection.engine
import cv2
from PIL import Image

def main():
    os.chdir('/home/pi/piCar/DeepPiCar/models/object_detection')

    parser = argparse.ArgumentParser()
    parser.add_argument(
      '--model', help='File path of Tflite model.', required=False)
    parser.add_argument(
      '--label', help='File path of label file.', required=False)
    args = parser.parse_args()

    args.model = 'data/model_result/mobilenet_ssd_v2_coco_quant_postprocess_edgetpu.tflite'
    args.label = 'data/model_result/coco_labels.txt'
```

Figura 45: Cambio de ruta en el *script*

Con la ruta cambiada, se ejecuta el *script* y aparece la salida de la cámara con la **Coral AI** ejecutando el modelo de reconocimiento de objetos del *dataset*[32] de **COCO** (figura 46).

COCO (*Common Objects in Context*) es un *dataset* que contiene una gran cantidad de imágenes de todo tipo, especialmente utilizado en visión artificial, tanto imágenes normales con etiquetas como con segmentación.

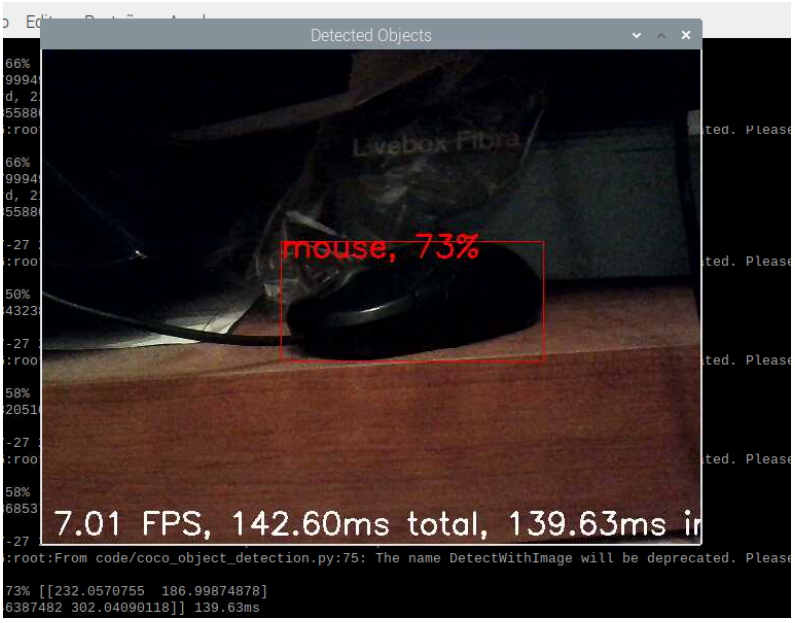


Figura 46: Prueba de la **Coral AI** utilizando **COCO**

Los pasos de la instalación de **PyCoral** se pueden encontrar en la web oficial[43].

5.3 Primera parte: detectar las líneas con el coche PiCar

Con todo instalado y funcionando, se puede pasar a crear un circuito para que el coche PiCar pueda recorrerlo. Primero lo recorrerá detectando las líneas, realizando unos cálculos y girando acorde al resultado, y luego aprendiendo a recorrerlo utilizando *Deep Learning*. Es importante que el circuito tenga una diferencia considerable entre la carretera y las líneas, en este caso la carretera es negra y las líneas blancas.
Tenía dificultades al crear un circuito ya que el suelo de mi casa es de varias piedras de varios colores(blanco, marrón, grises sobre un suelo rojo oscuro), y encontrar un cinta o algún objeto para delimitar que tenga un color significativamente diferente es complicado, por lo que al final decidí coger una caja negra y cinta blanca.
También cabe destacar que no es recomendable hacer las curvas muy cerradas ya que al coche le va a costar más girar, sobre todo a grandes velocidades. Esto es posible modificarlo en el código y se mostrará más adelante.
Cuando se tenga un circuito creado, se puede utilizar el *script* **color_HUE.py** creado para obtener los valores mínimos y máximos de RGB que harán falta en un siguiente paso (figura 47). Es posible utilizar este mismo programa para comprobar que los colores utilizados para la carretera y las líneas sean suficientemente diferentes. Lo que se busca con el programa es ajustar las barras para encontrar la silueta de las líneas del circuito en blanco y el resto en negro. Lo importante es que desde la mitad de la imágen hacia abajo se diferencie perfectamente las líneas. La parte de arriba o incluso los lados de las líneas se pueden ignorar por ahora ya que luego es posible cubrir esas partes utilizando el *script* **hand_coded_lane_follower.py**.

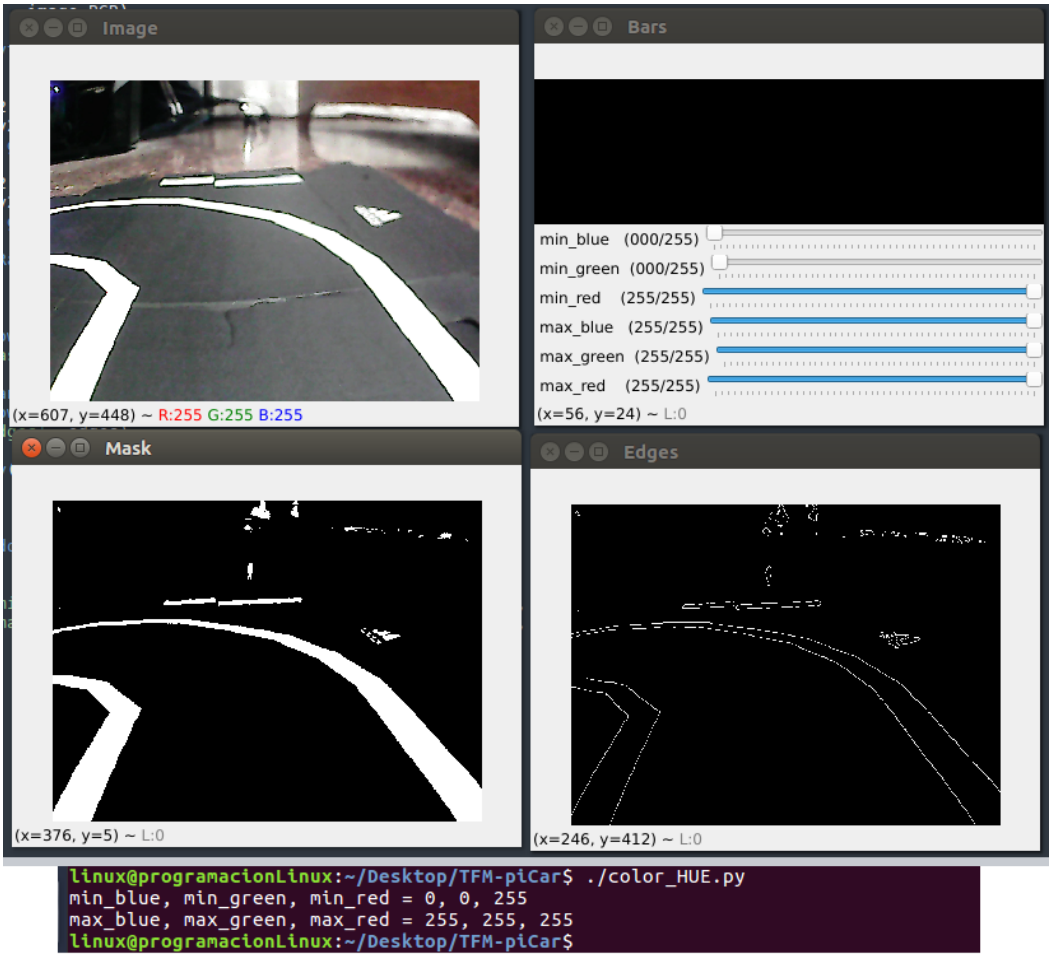


Figura 47: Vista del *script* **color_HUE.py** y su salida

Cuando se tenga las líneas definidas, con la tecla '*q*' se termina el programa y en la terminal saldrán los valores que hacen falta incluir en el código.

El objetivo es captar las líneas delimitantes del circuito con la cámara y calcular su silueta para conseguir tener dos líneas virtuales. Esto se consigue calculando las líneas más probables entre los puntos de la silueta utilizando la transformación de Hough (figura 48) y, a continuación, calcular la media de las pendientes de las líneas, diferenciando entre carril izquierdo y derecho.
La transformación de Hough[44] es una técnica que se utiliza para detectar todo tipo de formas si esas formas se pueden representar matemáticamente.

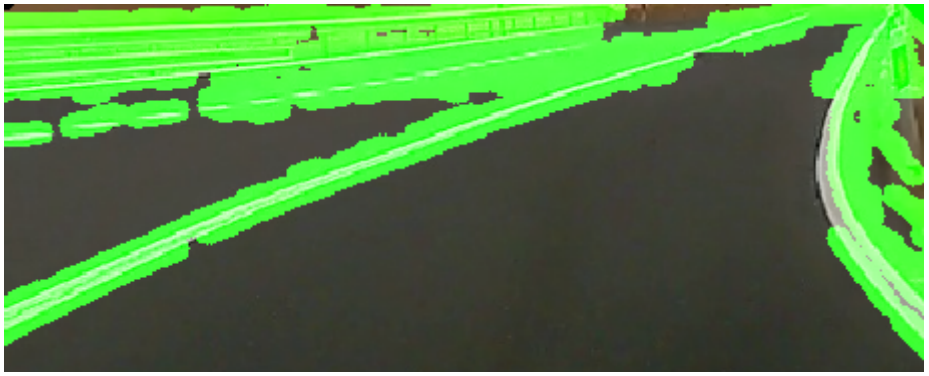


Figura 48: Transformación de Hough aplicada a una imágen de carretera

Las líneas empezarán en las esquinas inferiores a ambos lados, por lo que solo es necesario obtener un punto final, que es la media que se acaba de calcular. Ahora se calcula una tercera línea que será la encargada de indicar al coche hacía donde moverse a continuación, para esto simplemente calculamos la media de los dos puntos finales de las líneas calculadas anteriormente (figura 49). Está línea empezará siempre en la parte inferior central e igualmente que las líneas anteriores, es necesario un segundo punto final. Si solo se capta un carril, y por tanto una línea, el punto final de la línea guía se situará en paralelo a la única línea captada. Por último, se realiza una función para estabilizar los giros y que no se produzcan giros bruscos. Esto se consigue sumando o restando un valor fijo cada vez que el próximo valor de giro sea superior a un máximo establecido. El valor de giro del coche es un número entero entre 45 y 135 que indica cuánto se deben de mover las ruedas, siendo 90, el valor indicando que las ruedas están rectas, 45 girar al máximo a la izquierda y 135 al máximo a la derecha.

Figura 49: Líneas finales mostradas

Entendido esto, hay que situarse en la carpeta **~/DeepPiCar/driver/code** y modificar varios *scripts*, según nuestras preferencias. Primeramente **deep_pi_car.py**. Aquí se cambia el valor en la línea 134, este valor indica a qué velocidad se desplazará el coche. En este caso el valor es de 20 porque tras probar con el valor inicial de 40, el coche

avanzaba demasiado rápido para el circuito definido, y a pesar de que registraba correctamente cuando girar, no giraba con suficiente rapidez (figura 50).

```
132    def main():
133        with DeepPiCar() as car:
134            car.drive(20)
```

Figura 50: Líneas donde modificar la velocidad inicial

También es necesario añadir una línea, la línea 38 en la figura 51. Está línea indica que el coche debe avanzar, sin esto, el coche se queda estático. En el código original está instrucción se ha obviado por algún motivo (figura 51).

```
35
36          logging.debug('Set up back wheels')
37          self.back_wheels = picar.back_wheels.Back_Wheels()
38          self.back_wheels.speed = 0  # Speed Range is 0 (stop) - 100 (fastest)
39
```

```
35
36          logging.debug('Set up back wheels')
37          self.back_wheels = picar.back_wheels.Back_Wheels()
38          self.back_wheels.forward()
39          self.back_wheels.speed = 0  # Speed Range is 0 (stop) - 100 (fastest)
40
```

Figura 51: Añadir la línea de código

Ahora se edita el script **hand_coded_lane_follower.py**, donde en las líneas 77 y 78 se cambian los valores por los obtenidos anteriormente con el *script* **color_HUE.py** (figura 52).

```
73    def detect_edges(frame):
74        # filter for blue lane lines
75        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
76        show_image("hsv", hsv)
77        lower_blue = np.array([0, 0, 255])
78        upper_blue = np.array([0, 0, 255])
79        mask = cv2.inRange(hsv, lower_blue, upper_blue)
80        show_image("blue mask", mask)
81
82        # detect edges
83        edges = cv2.Canny(mask, 200, 400)
84
85        return edges
```

Figura 52: Cambiar los valores

La función **detect_line_segments()** detecta las líneas de la imágen con la función de Hough, y de la línea 129 del mismo *script*, se pueden modificar varios valores para tener mejores resultados a la hora de obtener las líneas virtuales que indican la delimitación del circuito (figura 53). Estos valores se encuentran con prueba y error, modificándolos y viendo si los resultados mejoran o empeoran.

```
129  def detect_line_segments(cropped_edges):
130      # tuning min_threshold, minLineLength, maxLineGap is a trial and error process by hand
131      rho = 3  # precision in pixel, i.e. 1 pixel
132      angle = np.pi / 180  # degree in radian, i.e. 1 degree
133      min_threshold = 10  # minimal of votes
134      line_segments = cv2.HoughLinesP(cropped_edges, rho, angle, min_threshold, np.array([]), minLineLength=10,
135                                      maxLineGap=4)
136
137      if line_segments is not None:
138          for line_segment in line_segments:
139              logging.debug('detected line_segment:')
140              logging.debug("%s of length %s" % (line_segment, length_of_line_segment(line_segment[0])))
141
142      return line_segments
```

Figura 53: Código de la función **detect_line_segments()**

La función **stabilize_steering_angle()** modifica el valor de giro para que los giros sean más suaves. En la línea 222, se pueden cambiar los valores de desviación cuando se tengan una o dos líneas para tener mejores giros. Con los parámetros iniciales de cinco y uno (figura 54), el coche **PiCar** tarda bastante en realizar los giros cuando se detecta una sola línea, resultando en su salida del circuito. Por ello el parámetro de desviación de una línea se cambió a ocho.

```
221
222     def stabilize_steering_angle(
223                         curr_steering_angle,
224                         new_steering_angle,
225                         num_of_lane_lines,
226                         max_angle_deviation_two_lines=5,
227                         max_angle_deviation_one_lane=1):
228
```

Figura 54: Valores a cambiar

Una vez esté todo cambiado, se comprueba que todo funcione correctamente, donde el coche **PiCar** debería de recorrer el circuito girando en las curvas marcadas con cinta.

5.4 Segunda parte: entrenar al coche PiCar para recorrer el circuito de forma autónoma

El proceso de entrenamiento y modelo utilizado está basado en un modelo realizado por **Nvidia** en el artículo "*End to End Learning for Self-Driving Cars*"[45]. Este modelo utiliza entrenamiento supervisado con regresión, utilizando los *frames* de vídeos grabados con una cámara desde un coche real, obteniendo los valores del ángulo de giro del volante. Se utiliza **CNN**, la red neuronal convolucional, para extraer los rasgos en cada imágen como se explicó en el punto ***2.2.1 Convolutional neural network (CNN) o red neuronal convolucional***. En este modelo, la imágen de entrada se normaliza, se pasa por cinco capas convolucionales y cuatro capas de neuronas totalmente conectadas produciendo como salida el ángulo de giro. También se utiliza propagación hacia atrás para corregir los errores en las predicciones.

Con esto, ahora se puede empezar a obtener los datos para entrenar el modelo del coche para que recorra el circuito sin ayuda. Con el script **save_training_data.py** se consiguen los *frames* del coche en movimiento cuando sigue las líneas. Se debe indicar

de qué vídeo obtener los *frames* por medio de la consola. Además de guardar cada *frame* (figura 55), también se añade un número al nombre de la imágen que contiene ese *frame*, que es el valor de giro de las ruedas delanteras (figura 56). De este modo un número menor de 90 significa que el coche está girando a la izquierda, y mayor de 90 a la derecha.

```
pi@raspberrypi:~/tmp_2/DeepPiCar/driver/data/tmp $ ls *.png
car_video220428_215004_000_085.png  car_video220428_215004_009_059.png  car_video220428_215004_018_065.png  car_video220428_215004_027_056.png
car_video220428_215004_001_080.png  car_video220428_215004_010_058.png  car_video220428_215004_019_062.png  car_video220428_215004_028_061.png
car_video220428_215004_002_080.png  car_video220428_215004_011_056.png  car_video220428_215004_020_054.png  car_video220428_215004_029_057.png
car_video220428_215004_003_080.png  car_video220428_215004_012_056.png  car_video220428_215004_021_059.png  car_video220428_215004_030_060.png
car_video220428_215004_004_072.png  car_video220428_215004_013_060.png  car_video220428_215004_022_056.png  car_video220428_215004_031_052.png
car_video220428_215004_005_064.png  car_video220428_215004_014_065.png  car_video220428_215004_023_061.png  car_video220428_215004_032_055.png
car_video220428_215004_006_056.png  car_video220428_215004_015_057.png  car_video220428_215004_024_061.png  car_video220528_205926_000_090.png
car_video220428_215004_007_055.png  car_video220428_215004_016_056.png  car_video220428_215004_025_058.png
car_video220428_215004_008_054.png  car_video220428_215004_017_060.png  car_video220428_215004_026_055.png
pi@raspberrypi:~/tmp_2/DeepPiCar/driver/data/tmp $
```

Figura 55: Directorio con los *frames* guardados con los valores de giro

```
car_video220428_215004_000_085.png
```

Figura 56: Nombre de un *frame* con la fecha, hora, el número del *frame* y el valor de giro

Se recorre el circuito un par de veces, y con los *frames* guardados, descartamos los *frames* erróneos que hayan podido surgir. Por ejemplo, en un *frame* durante una curva a la izquierda se indica que gire a la izquierda con un valor de giro de 75, pero en los siguientes *frames* no se ha podido detectar correctamente las líneas y se indica un valor de giro de 80 y 85 en cada *frame*. Estos dos *frames* nos darán problemas a la hora de entrenar el modelo ya que le estamos indicando que durante una curva, cuanto más cerca de la línea, el coche debe de girar menos, produciendo que el coche **PiCar** se salga del circuito.
En el ejemplo de la figura 57 se observa claramente como el valor de giro debe ser mayor de 90, pero el resultado indica que el valor es de 73, por lo que este *frame* se puede eliminar.

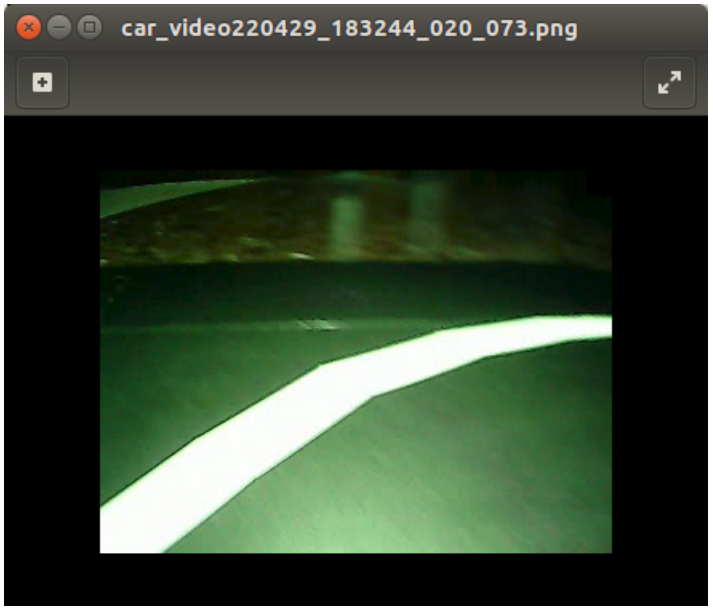


Figura 57: Ejemplo de *frame* erróneo

Para realizar el entrenamiento del modelo se utilizará **Colab**. Con **Colab** no hay necesidad de configurar nada fuera de los módulos que necesitemos utilizar. **TensorFlow** ya está instalado y configurado, por lo que con importarlo se ahorran las horas de instalación y configuración si es necesario. Dispone además de una buena GPU y es gratuito.
Es posible entrenar el modelo sin necesidad de usar **Colab**, pero se requiere un ordenador con una GPU potente. La parte negativa de utilizar **Colab** de forma gratuita es que se requiere conexión a Internet para utilizar y tiene un límite de uso, que una vez superado, no se puede conectar a la GPU durante un día. Además, si se supera de nuevo el límite al día siguiente, el periodo sin poder conectarse a la GPU se extiende. Los límites de uso de **Colab** y periodos de desconexión no parece que sean de dominio público por lo que es recomendable utilizarlo lo necesario.

Cuando se tenga suficientes imágenes correctas que se ajusten a como debería de recorrer el circuito si estuviera siendo conducido por un conductor, se separan las imágenes en imágenes de entrenamiento (*training*) y de prueba (*test*). Normalmente es un 80% para entrenar y un 20% para probar. Se guardan las imágenes en una carpeta para *training* y otra para *test*, y se suben las dos carpetas a **Drive**.

Se empieza enlazando el **Drive** con **Colab** para poder acceder a las imágenes que se han subido, y se instala de nuevo el módulo **h5py** a una versión anterior porque con la actual se tiene un problema accediendo a los vector de otro módulo, **numpy**. Se importan los diferentes módulos, teniendo en cuenta que se debe indicar que se quiere utilizar la versión 1 de **TensorFlow**. Como se comentó en el punto ***3.4 Plan de trabajo***, esta versión ya no está soportada y, por tanto, se debe utilizar la versión 2 si se quiere utilizar la GPU de **Colab** y agilizar el entrenamiento.
En este caso, el modelo inicialmente se realizó con **TensorFlow** versión 1, pero para poder utilizarse actualmente después de los cambios, se requiere desinstalar **TensorFlow** e instalar la versión 1.5.2. Con esto se puede entrenar el modelo pero el tiempo en finalizar será superior ya que se está utilizando la CPU en vez de GPU. Para utilizar la GPU es necesario instalar una serie de módulos y su instalación y configuración es complicada y larga, por lo que se requiere realizar estos pasos, lo mejor es hacerlo en una máquina local.
Esto es obligatorio realizar porque la versión más reciente de **TensorFlow** que puede utilizarse en **Raspberry Pi** es la versión 1.5, si se entrena el modelo en alguna versión superior y luego se ejecuta en **Raspberry Pi** con la versión 1.5, aparecen errores y no se puede ejecutar.

A continuación se cargan las imágenes y se crea un *DataFrame*[33] con la ruta de cada imágen y su ángulo de giro, y se generan varios histogramas[34] para ver el número de ángulos de giro que se tienen y poder añadir más si es necesario.
Se añaden varias funciones que nos permitirán aumentar el número de imágenes para el entrenamiento. Esto se denomina *Data augmentation* y es una técnica utilizada para tener más imágenes de las producidas de una forma rápida y sencilla. Las funciones

implementadas son: una función de *zoom* de la imágen, **zoom()**, la función **pan()** que mueve la imágen una cantidad aleatoria de milímetros, una función **adjunst_brightness()** que cambia el brillo de la imágen, otra función **blur()** que difumina un poco la imágen, y otra función que gira horizontalmente la imágen produciendo un efecto espejo. Hay que tener en cuenta que en esta función también se debe cambiar el valor del ángulo de giro a su inversa. Todas estas funciones se aplican aleatoriamente a cada imágen que se tenga en las carpetas.
En la función **nvidia_model()** se implementa el modelo utilizado por **Nvidia** mencionado anteriormente, se crea la función auxiliar **image_data_generator()** que permite crear imágenes dinámicamente de las que se tienen en la carpeta de *training* y utilizando las funciones de transformación y, por último, se realiza el entrenamiento.

Cuando finalice la penúltima celda del código de **Colab**, se tendrá en el **Drive** un fichero que contiene el modelo entrenado, se descarga y copia a la **Raspberry Pi** en la carpeta **~/DeepPiCar/code/models**.
Se puede comprobar que no se ha producido sobreajuste, sacando la gráfica de aprendizaje (figura 58).

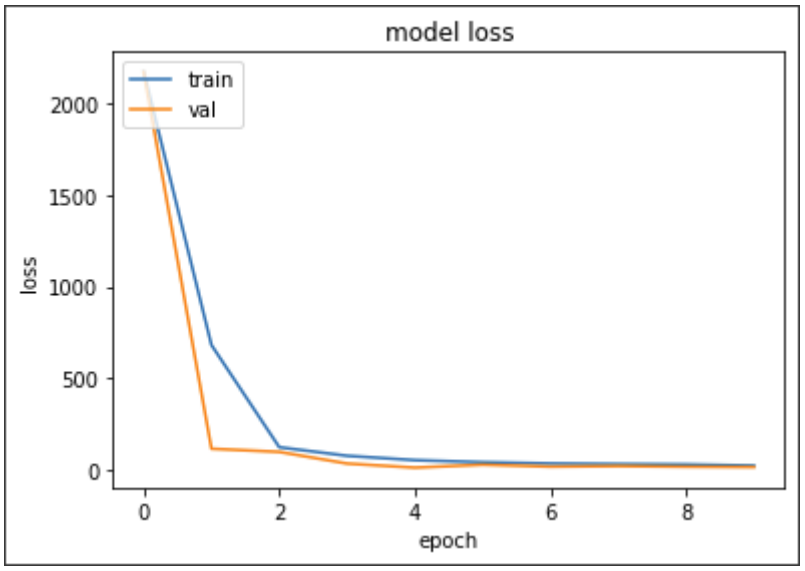


Figura 58: Gráfica de aprendizaje

La última celda se puede realizar para ver la precisión del modelo, un valor cercano al 100% es buena señal.
En **~/DeepPiCar/driver/code** se localiza el *script* **deep_pi_car_dl.py** que se lanzará para ver si todo funciona correctamente. Es posible que aparezca un error al ejecutar el *script* como se muestra en la figura 59. Para solucionarlo, se debe ejecutar el siguiente comando parecido al que se realizó en **Colab**:

```
pip3 install 'h5py==2.10.0' --force-reinstall
```

```
Traceback (most recent call last):
  File "arrowCarModel.py", line 52, in <module>
    model = load_model(model_path, compile = False)
  File "/home/pi/.local/lib/python3.7/site-packages/tensorflow_core/python/keras/saving/save.py", line 146, in load_model
    return hdf5_format.load_model_from_hdf5(filepath, custom_objects, compile)
  File "/home/pi/.local/lib/python3.7/site-packages/tensorflow_core/python/keras/saving/hdf5_format.py", line 166, in load_model_from_hdf5
    model_config = json.loads(model_config.decode('utf-8'))
AttributeError: 'str' object has no attribute 'decode'
```

Figura 59: Posible error al ejecutar el *script*

En este *script* se hace una llamada al *script* **end_to_end_lane_follower.py**, que es el encargado de realizar la predicción para indicar hacía dónde dirigir al coche **PiCar**. Es importante indicar la ruta correcta del modelo entrenado en la línea 14 como se indica en la figura 60.

```
11
12      def __init__(self,
13                   car=None,
14                   model_path='/home/pi/tmp_2/DeepPiCar/driver/code/models/lane_navigation_final.h5'):
15          logging.info('Creating a EndToEndLaneFollower...')
16
```

Figura 60: Modificar la ruta donde se encuentre el modelo

5.5 Tercera parte: reconocimiento de señales

Terminado el proceso de entrenamiento y la comprobación de que el coche **PiCar** recorre correctamente el circuito esta vez con *Deep Learning*, se puede empezar a entrenar otro modelo para que detecte las diferentes señales de circulación y reaccione acordemente. Se empieza creando las señales que queremos tener, en este caso una señal de "STOP" donde el coche deberá de detenerse y, cuando se quite la señal, el coche continúe; un semáforo en verde y otro en rojo, si detecta el semáforo en rojo deberá de detenerse hasta que detecte un semáforo en verde, incluso quitando el semáforo rojo, el coche deberá esperar; y dos señales de velocidad, una de 10 y otra de 30, que cambiarán a que velocidad debe moverse el coche. Una vez creadas, se usará la cámara conectada al coche **PiCar** para obtener imágenes de las diferentes señales. Se puede usar cualquier método para obtener las imágenes, como por ejemplo usar el programa para guardar los *frames* del circuito pero enfocando las señales y luego eliminar las imágenes que no contengan señales o sean de mala calidad. Otra forma, que es la utilizada, es crear un programa que guarde el *frame* actual cada vez que se presione un botón. El script **sign_frame_save.py** funciona presionando el botón de la barra espaciadora para guardar un único *frame*. Es importante que las imágenes de las señales sean diferentes y que contengan más de una señal, que estén en diferentes escenarios y se puedan diferenciar entre ellas, como por ejemplo las imágenes mostradas en las figura 61.

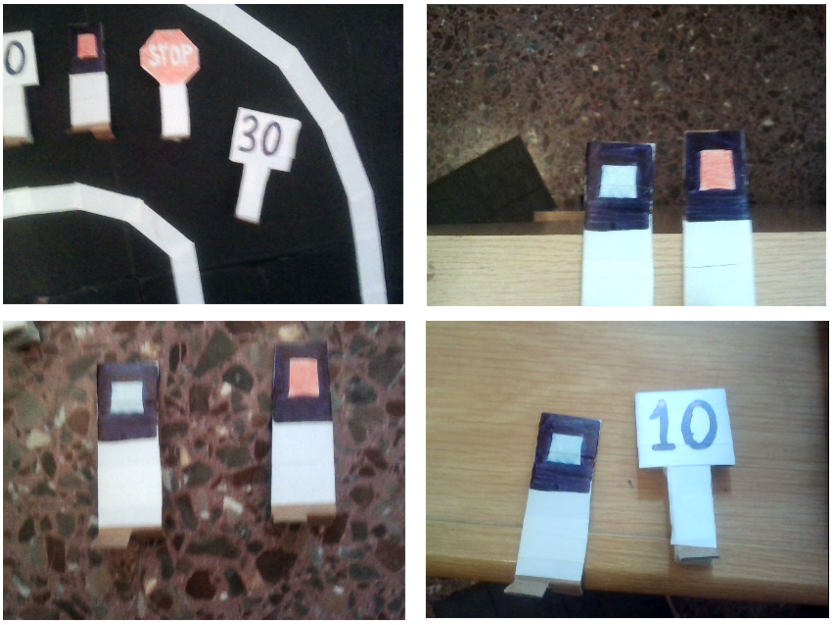


Figura 61: Diferentes posiciones de las señales

Se toman unas 50 imágenes, como se recomienda. Obtenidas las imágenes, se procede a instalar el programa **LabelImg**[46] que permitirá crear etiquetas para cada señal que hace falta para entrenar el nuevo modelo.
Se puede instalar con el comando `pip3 install labelImg` en **Linux**, o consultando el **Github** para los demás sistemas operativos. En **Linux**, se abre una terminal y se lanza el programa escribiendo `labelImg` (figura 62).

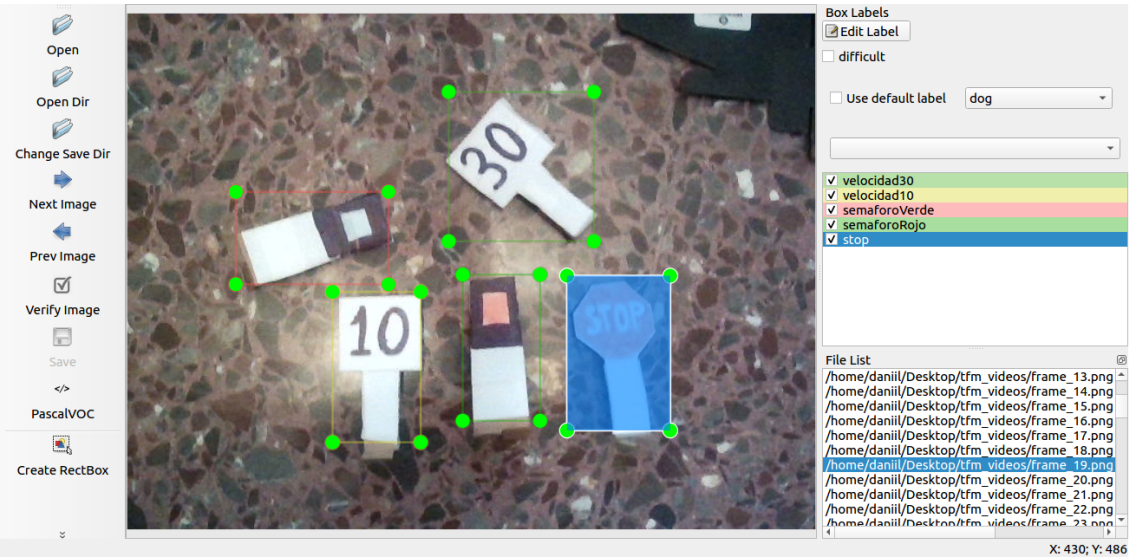


Figura 62: Programa **LabelImg**

En la parte izquierda se abre el directorio donde se tengan las imágenes de las señales con "*Open Dir*", y se cambia el directorio donde se guardaran los ficheros con las etiquetas con "*Change Save Dir*". Con las flechas "*Next Image*" y "*Prev Image*" es posible moverse de una imágen a otra, también es posible moverse a cualquier imágen seleccionandola en la lista de imágenes de la parte derecha. El botón de "*Save*" permite

guardar las etiquetas que se hayan creado en la imágen y generar el fichero con dichas etiquetas. **LabelImg** soporta tres convenciones para guardar las etiquetas. **PascalVOC** produce el fichero resultante en formato *.xml*, que contiene una amplia lista de características como nombre de la imágen, su ruta donde se encuentre o las dimensiones de la etiqueta y su nombre. **YOLO** produce un fichero con formato *.txt*, el cual solo contiene el número identificador de la etiqueta y la posición de la etiqueta calculada en porcentaje referente a la imágen. Finalmente, **CreateML** genera un fichero *.json*, que contiene el nombre de la imágen, el nombre de la etiqueta y las coordenadas de cada una de las etiquetas en pixeles. Para el modelo utilizado, se requiere que esté en formato *.xml*, por lo que se selecciona la opción **PascalVOC**.
En la parte derecha se tendrá una lista de imágenes, que serán las que se tengan en el directorio abierto, y justo en la parte superior, una lista de las etiquetas que tiene la imágen. Inicialmente no se tendrá ninguna, pero a medida que se añaden etiquetas, irán apareciendo (figura 63).

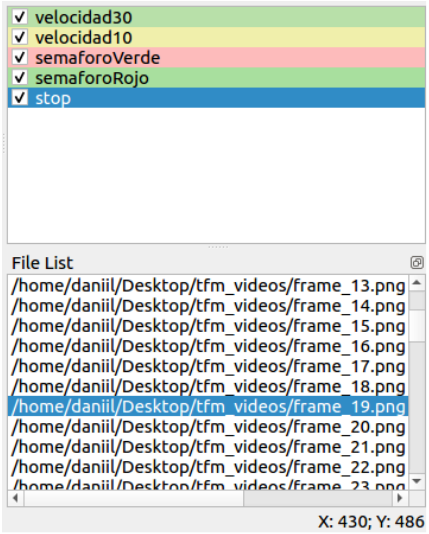


Figura 63: Etiquetas y ficheros del directorio

Para crear una etiqueta se selecciona "*Create RectBox*" del menú de la parte izquierda. Ahora se marca un cuadrado o rectángulo abarcando toda la forma que se quiere etiquetar, se le pone un nombre identificador, y aparecerá en la parte derecha encima de la lista de imágenes.
Se repite este proceso con todas las imágenes, teniendo en cuenta que cuanto más precisas las formas de la etiqueta, mejor detectará los objetos con esa forma en el modelo final. //explayarme mas

Terminado el proceso de etiquetar, se crean tres carpetas, *test*, *training* y *validation*. Dentro de cada una de las carpetas, se crea otras dos, *images* y *annotations*. Ahora se separan las imágenes en 70% *training*, 20% *test* y 10% *validation* como se realizó anteriormente con el primer modelo a excepción de *validation*, y se mueven a su correspondiente carpeta, sin dejar los ficheros *.xml*. Si se han usado 50 imágenes, se debería de tener 35 ficheros en **/training/images**, otros 35 en **/training/annotation**, 10

en **/test/images**, 10 en **/test/annotations**, 5 en **/validation/images** y 5 en **/validation/annotations**. A continuación se suben las tres carpetas con las imágenes y *.xml* a Drive y se procede a entrenar el nuevo modelo con las señales.
Las imágenes en la carpeta de *Validation*, o validación, se usan en este caso para afinar los parámetros de las capas y comprobar que el modelo que se está entrenando tiene un sobreajuste. A diferencia de las imágenes en la carpeta de *test* que solamente se utilizan para comprobar el modelo final.

Antes de comenzar, se debe elegir el modelo que sirva. Coral AI está diseñada para utilizar un modelo *quantized*, como se explicó anteriormente, los valores de los parámetros utilizados deben estar definidos como valores enteros en vez de como coma flotante como lo están la gran mayoría de modelos normales. Además, se tiene que tener en cuenta que el tamaño del modelo no exceda los 8 MB si se quiere que no tenga mucha latencia, y en caso de que supere los 8 MB, se tendrá que particionar el modelo o utilizar más de un acelerador USB.
El proceso para entrenar el modelo explicado en el artículo no se consiguió realizar, al estar escrito hace tres años, se han cambiado muchas cosas, sobretodo los módulos **TensorFlow** y **NumPy**, por lo que requiere varios cambios al principio de la ejecución para que funcione correctamente.
A continuación se indican los cambios realizados para que funcione, pero no se utilizará este modelo, el utilizado se explicará más adelante.

Lo primero es instalar los módulos indicados en la figura 64, teniendo en cuenta que se deben de desinstalar los módulos **NumPy** y **pycocotools** para instalar versiones anteriores.

```
[ ] !python --version
    Python 3.7.13

[ ] !pip uninstall -y numpy

[ ] !pip install numpy==1.19

[ ] !pip uninstall -y pycocotools

[ ] !pip install pycocotools --no-binary pycocotools

[ ] !pip install tf_slim

[ ] !pip install absl-py

[ ] !pip install lvis

[ ] %tensorflow_version 1.x
    import tensorflow as tf
```

Figura 64: Módulos instalados

Una vez instalados estos módulos, es posible continuar con el proceso descrito. Pero al llegar al punto de entrenar el modelo, después de varias iteraciones, cuando muestra los detalles del proceso del entrenamiento, la precisión aparece a cero (figura 65), indicando que algo no va bien ya que debería de haber algún número.

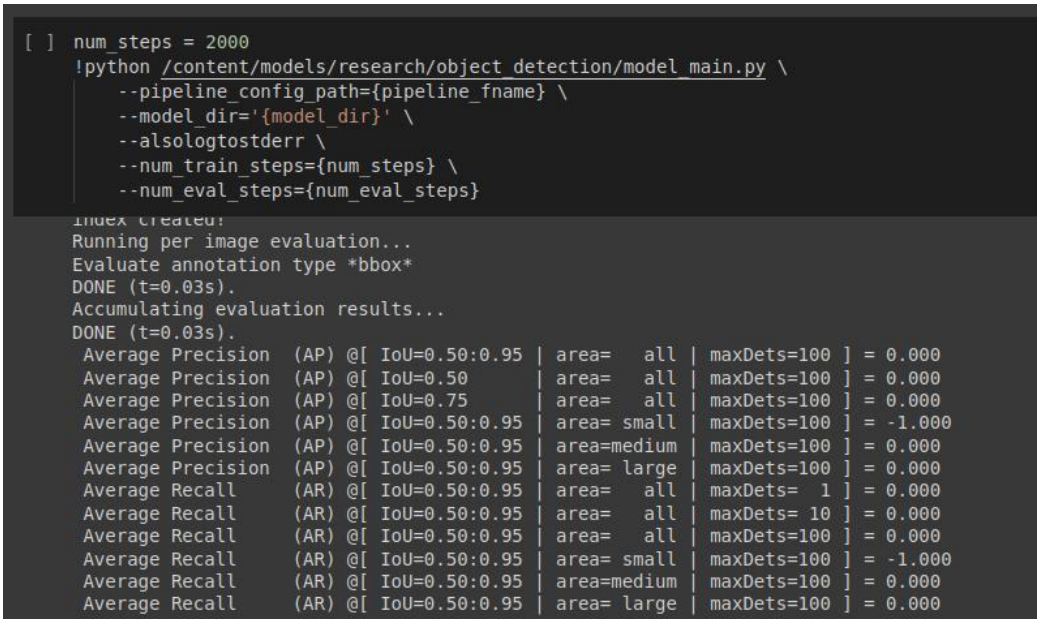


Figura 65: Salida del paso del entrenamiento

Lo más común, cuando se producen estos tipos de errores, es que el problema reside en que las rutas no están correctamente definidas, no se encuentra las imágenes o hay algún problema al leer las imágenes, pero después de muchas pruebas e intentos, no se consiguió solucionar el problema, todo parece estar correctamente definido y configurado, tanto con las rutas como las imágenes.

A partir de aquí se buscó una alternativa, y el método utilizado que sí funciona y se puede reproducir, está basado en una guía proporcionada por **Coral** en la sección de "Documentación"[47] utilizando **EfficientDet-Lite**. En la misma sección de "Documentación", podemos encontrar una serie de modelos que funcionan con la **Coral**, por lo que se utilizará el modelo **EfficientDet-Lite1** ya que tiene la mayor precisión y ocupa menos de 8MB.

Se empieza instalando las bibliotecas necesarias. El comando indicado en la guía no es correcto, ya que el proceso de instalación se mantiene durante horas y no se completa nunca, resultando en **Colab** desconectando la sesión. Esto es debido a los últimos cambios realizados en **Colab** como se comentó en el punto ***3.4 Plan de trabajo*****,** por lo que se debe utilizar los siguientes comandos:

```
!git clone https://github.com/tensorflow/examples
%cd examples/tensorflow_examples/lite/model_maker/pip_package
!pip install -e .
```

A continuación se importan los módulos a utilizar y se enlaza **Colab** con **Drive**. Ahora se pueden subir las imágenes que contienen las señales y las anotaciones de las mismas. Pero primero se deben crear tres carpetas: una de *test*, una de *training* y una de validación en **Drive**, y dentro de cada carpeta se crean otras dos carpetas, una para las imágenes y otra para las anotaciones. El resultado debe ser parecido a la figura 66:

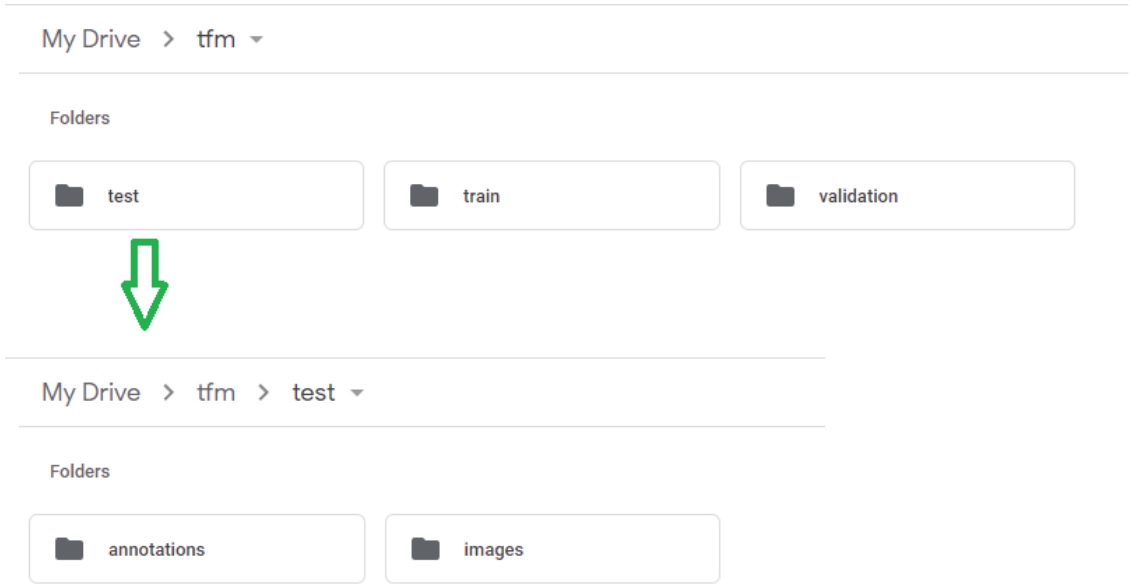


Figura 66: Carpetas finales en **Drive**

En esta ocasión, a diferencia de los puntos anteriores, las imágenes de *training* serán un 70% mientras que la validación será de un 10%. Estos valores se pueden cambiar a un 80% y un 10% para *test* y validación, o valores similares, pero es recomendable que se tenga una gran cantidad de imágenes de entrenamiento.
A continuación se indican las etiquetas que se tengan en las imágenes, en este caso se tiene una señal de stop, dos semáforos, uno verde y uno rojo, y dos señales de velocidad, una de 10 y otra de 30. Tener en cuenta que se debe empezar por el número 1 al indicar las etiquetas, ya que el 0 está reservado. También se indican las rutas donde se encuentran las imágenes y las anotaciones y se cargan las imágenes y las anotaciones teniendo en cuenta las etiquetas indicadas anteriormente.
Luego se especifica el modelo a utilizar con el comando:

```
spec = object_detector.EfficientDetLite1Spec()
```

Y se comienza a entrenar el nuevo modelo. Dependiendo de las imágenes que se tengan, este proceso puede tardar más o menos tiempo. Con pocas imágenes el proceso puede ser de minutos. Se puede cambiar el valor de *epoch* y *batch_size* si se conviene. Estos valores hacen referencia a la cantidad de iteraciones a realizar y el tamaño de cada lote de imágenes, respectivamente.
En esta parte, a la hora de elegir el modelo a utilizar, hay que recordar que si se utiliza un modelo superior a 8MB de tamaño, como es el caso de **EfficientDet-Lite2** y **EfficientDet-Lite3**, se tendrá una latencia superior de lo normal, por lo que se debería utilizar **EfficientDet-Lite0** o **EfficientDet-Lite1** si solo se tiene un acelerador USB como es el caso de este TFM.

Una vez el proceso de entrenamiento termine, se puede exportar el nuevo modelo para que se pueda utilizar con **TensorFlow Lite** y se prueba el modelo utilizando **PyCoral API**. La función **export()** utilizada, sirve para exportar el modelo para ser utilizado con **TensorFlow Lite** y, además, realiza cuantización de números enteros durante el mismo proceso, lo cual es algo beneficioso para poder ser utilizado en una **Edge TPU**, la **Coral** en este caso.

Con el modelo probado, se debe compilar el modelo para utilizarse en la Coral. Se instala el compilador para **Edge TPU** y se ejecuta el comando:

```
!edgetpu_compiler
'/content/drive/MyDrive/tfm/efficientdet-lite-signs-eDet1.tflit' -d --num_segments=$NUMBER_OF_TPUS
```

Donde se debe indicar la ruta donde se guardará el modelo compilado y el número de TPUs a utilizar (una en este caso, `NUMBER_OF_TPUS = 1`)
Finalizado el proceso, se descarga el modelo de **Drive** y se guarda en la carpeta **~/DeepPiCar/code/models**, donde se encuentra en modelo anteriormente creado.

Como se realizó anteriormente, para ver si hay sobreajuste se obtiene la gráfica de aprendizaje (figura 67).

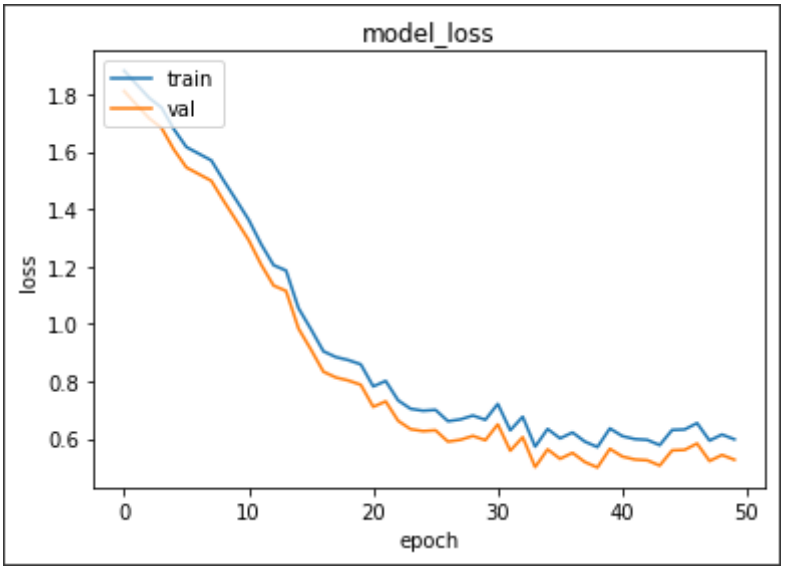


Figura 67: Gráfica de aprendizaje

Antes de poder probar el modelo creado, es necesario cambiar los scripts originales y adaptarlos para que utilicen las nuevas llamadas de **PyCoral API**.
Primeramente se deben de importar varios módulos de **PyCoral** como se indica en la figura 68.

```
 7
 8  #import edgetpu.detection.engine
 9  from pycoral.adapters import common
10  from pycoral.adapters import detect
11  from pycoral.adapters.common import input_size
12  from pycoral.utils.dataset import read_label_file
13  from pycoral.utils.edgetpu import make_interpreter
14  from pycoral.utils.edgetpu import run_inference
15
```

Figura 68: Módulos a importar

Luego se cambian las líneas 41 a 46 del *script* original a las mostradas en la figura 69.

```
44
45          self.labels = read_label_file(label)
46          logging.info('Initialize Edge TPU with model %s...' % model)
47          self.interpreter = make_interpreter(model)
48          self.interpreter.allocate_tensors()
49          #self.interpreter.invoke()
50          self.inference_size = input_size(self.interpreter)
51          logging.info('Initialize Edge TPU with model done.')
52
```

Figura 69: Instrucciones nuevas

También es importante cambiar la línea 67 (figura 70) y poner los nombres de los objetos que se tenga en el fichero **classes.txt** en el mismo orden que aparecen. Está función es la que se encarga de dar las diferentes instrucciones cuando se detecta un objeto u otro.

```
66
67          self.traffic_objects = {0: GreenTrafficLight(),
68                                  1: Person(),
69                                  2: RedTrafficLight(),
70                                  3: SpeedLimit(25),
71                                  4: SpeedLimit(40),
72                                  5: StopSign()}
73
```

```
64
65          self.traffic_objects = {0: StopSign(),
66                                  1: SpeedLimit(30),
67                                  2: SpeedLimit(10),
68                                  3: RedTrafficLight(),
69                                  4: GreenTrafficLight()}
70
```

Figura 70: Cambio de los nombres de los objetos que se quieren detectar

La función **detect_objects()** del *script* original se debe cambiar casi por completo. Empezando por preparar la imágen y luego llamando a la función **get_objects()** de **PyCoral** para que realice las predicciones con la **Coral** (figura 71).

```
149
150         cv2_im = frame
151         cv2_im_rgb = cv2.cvtColor(cv2_im, cv2.COLOR_BGR2RGB)
152         cv2_im_rgb = cv2.resize(cv2_im_rgb, self.inference_size)
153
154         run_inference(self.interpreter, cv2_im_rgb.tobytes())
155
156         objects = detect.get_objects(self.interpreter, self.min_confidence)[:self.num_of_objects]
157
158         if objects:
159             frame = self.append_objs_to_img(cv2_im, self.inference_size, objects, self.labels)
160         else:
161             logging.debug('No object detected')
162
```

Figura 71: Preparado de imágen y llamada a función

En caso de detectar algún objeto, se ejecuta la función **append_objs_to_img()** la cual se encarga de obtener los valores de X e Y y crear un rectángulo con el nombre del objeto detectado para indicar dicho objeto en la imágen (figura 72).

```
188
189     def append_objs_to_img(self, cv2_im, inference_size, objs, labels):
190         height, width, channels = cv2_im.shape
191         scale_x, scale_y = width / inference_size[0], height / inference_size[1]
192         for obj in objs:
193             bbox = obj.bbox.scale(scale_x, scale_y)
194             x0, y0 = int(bbox.xmin), int(bbox.ymin)
195             x1, y1 = int(bbox.xmax), int(bbox.ymax)
196
197             percent = int(100 * obj.score)
198             label = '{}% {}'.format(percent, labels.get(obj.id, obj.id))
199
200             cv2_im = cv2.rectangle(cv2_im, (x0, y0), (x1, y1), (0, 255, 0), 2)
201             cv2_im = cv2.putText(cv2_im, label, (x0, y0+30),
202                                  cv2.FONT_HERSHEY_SIMPLEX, 1.0, (255, 0, 0), 2)
203         return cv2_im
204
```

Figura 72: Función **append_objs_to_img()**

Con estos cambios, ahora es posible ejecutar el *script* inicial y comprobar que se detectan correctamente las señales.

5.6 Adaptación de lo realizado a un coche real

Teniendo el coche **PiCar** recorriendo un circuito, detectando y respetando las diferentes señales sin necesidad de interacción con el usuario es interesante, pero no tiene un uso práctico en el día a día. La opción más directa sería crear un modelo que se pueda utilizar, por ejemplo, mientras se conduce un coche real, donde se detecten las líneas de la carretera intentando mantenerse dentro de ellas, y además respetando las diferentes señales que existan.
Para conseguir esto, se utilizarán los mismos métodos usados en la creación de los modelos del coche **PiCar**, empezando por crear un modelo para circular por la carretera y más tarde un modelo para detectar las señales e interactuar acordemente con ellas.

Siguiendo lo realizado anteriormente con el coche **PiCar**, primeramente es necesario obtener las imágenes de la carretera y los ángulos de giro. Este proceso puede ser muy simple si se tiene un sistema **EPS** (*Electric Power Steering*) donde se podría obtener el ángulo de giro.
EPS es un sistema de dirección asistida eléctrico, que consta de de un motor eléctrico que ayuda en la conducción por medio de sensores posicionados en la columna de dirección que detectan el momento y, por medio de un microordenador, ajustar y asistir al conductor dependiendo de las condiciones actuales de la carretera y del vehículo.
Al tener un microordenador instalado, sería posible acceder a los datos utilizados por dicho microordenador, y obtener información relacionada con el giro del volante e incluso la velocidad actual del vehículo.
Sin embargo, estos sistemas no se encuentran en todos los coches. Cada vez más se está instalando este tipo de sistemas en coches más modernos, pero en mi caso, mi coche no dispone de este tipo de sistema, por lo que se debe encontrar otra forma de obtener el valor de giro.
Para conseguir esto, se crea un nuevo *script* tomando código del *script* **hand_coded_lane_follower.py**, que se utilizó anteriormente para crear el modelo del coche **PiCar** para que recorra el circuito de forma autónoma, pero en la función **region_of_interest()**, se indica la zona donde se quiere conseguir el contorno de las líneas, la zona siendo la parte de la carretera que se tenga enfrente. De esta forma, si se detecta cualquier otra cosa a los lados de la carretera, arriba de la misma o el salpicadero o panel de instrumentos del coche, no se tendrá en cuenta a la hora de crear las líneas virtuales para conseguir el valor de giro posteriormente.

El primer paso es obtener una imágen donde se esté avanzando en línea recta, el valor de giro en este caso será de 90, como en el modelo del coche **PiCar**. Este valor de giro se puede modificar y utilizar otros valores según convenga, pero para utilizar los mismos recursos y métodos que los utilizados con el coche **PiCar**, se seguirá el mismo formato.
Con la imágen obtenida, se busca el contorno de las líneas como se realizó anteriormente con el *script* **color_HUE.py**, a continuación se escriben estos valores en

el *script* **detect_lines_road.py** y se ejecuta dicho *script* indicando un vídeo de entrada, que es el vídeo grabado con la cámara de la carretera que se recorre previamente, y un directorio de salida, que será donde se guarden las imágenes del vídeo con su correspondiente valor de giro.
Para obtener el vídeo de la carretera, se utilizó el *script* **road_frame_save.py** donde se consigue la señal de una cámara conectada a un ordenador portátil en este caso, y donde se presiona la barra espaciadora cada vez que se quiera empezar y parar de guardar *frames*.

Con las imágenes obtenidas, se revisará cada imágen eliminando aquellas que su valor de giro no corresponda con lo que debería, de la misma forma que se realizó con el modelo del coche **PiCar**. También es aconsejable modificar manualmente cualquier valor giro que se vea necesario. Por ejemplo, si se circula en línea recta, el valor debería de ser 90 o parecido, si aparece un valor de 100 o superior, o 80 o inferior, se debería eliminar esta imágen, o modificar manualmente el valor editando el nombre de la imágen, para reducir en la mayor medida los errores del modelo final. Este tipo de errores no se producirían con un sistema EPS, ya que los valores serían precisos.

Si se observan las imágenes obtenidas, se puede apreciar como hay una gran cantidad de espacio sin carretera, y por tanto, es información inservible y que produciría errores a la hora de detectar la carretera en el modelo final. Esto es debido a que además de las características de la carretera, también se tendrían características del entorno, y eso no es necesario. Para evitar esto y que aparezca lo máximo posible la carretera y tener más precisión en el modelo final, se debe recortar la parte superior de la carretera en la imágen (figura 73). Pero si se utiliza el mismo código para entrenar el modelo para el coche **PiCar**, este paso no es necesario ya que en el código de **Colab** se recorta la imágen por la mitad en la función **img_preprocess()**.

Figura 73: Imágen recortada

Ahora se pueden subir las imágenes a **Drive** y realizar los mismos pasos que con el coche **PiCar**. Normalmente se separarían las imágenes en prueba y entrenamiento, con un 20% y un 80% de las imágenes totales respectivamente, pero la función **train_test_split()** del módulo **sklearn** lo hace por sí mismo.
A continuación ejecutar las celdas de **Colab** y descargar el modelo resultante.

Se obtiene también su gráfica de aprendizaje (figura 74).

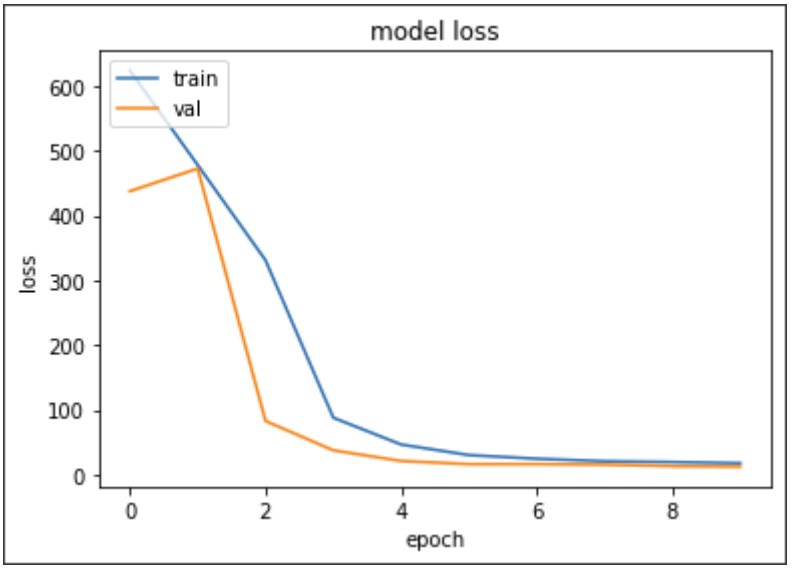


Figura 74: Gráfica de aprendizaje

Para probar que el modelo es correcto, se crea otro *script* llamado **arrowCarModel.py**. En este caso se obtiene las imágenes de la cámara o de un vídeo y en función de lo detectado por el modelo, se muestra una flecha en la imágen indicando hacía donde se debe girar si es necesario, o avanzar en línea recta. Esto se hace ya que no hay forma de poder interactuar directamente con el coche, si hubiese forma, se podría proporcionar el valor de giro del modelo directamente al coche y que este gire automáticamente sin interacción del usuario.

Con el correcto funcionamiento del modelo, se puede comenzar a crear un modelo que detecte las señales de tráfico.
Como se realizó anteriormente, se recogen imágenes de señales reales para realizar otro modelo. Para este caso es posible conseguir las imágenes de Internet o incluso el modelo en sí, ya que este tipo de aplicaciones de inteligencia artificial es bastante común, pero teniendo una forma de realizar el modelo, se opta por recoger las imágenes y crear el modelo.
Con las imágenes obtenidas, se consiguen las etiquetas utilizando el programa **LabelImg** de la misma forma que se realizó antes. Con las etiquetas e imágenes, se dividen en *training*, *test* y *validation*, y se suben a **Drive** repartidas en las diferentes carpetas.

Está vez en el código de **Colab**, se cambian algunas rutas para no sobreescribir el modelo realizado anteriormente y se modifican los valores de *epochs* y *batch_size* para aumentar la precisión del modelo al tener que realizar el entrenamiento más veces y con más imágenes. Se entrena el modelo, se obtiene el gráfico de aprendizaje (figura 75), se compila y se guarda en la carpeta correspondiente del coche **PiCar**. En realidad el coche **PiCar** ya no es necesario, solamente la **Raspberry Pi** junto a la **Coral**, pero las

baterías están integradas en el coche **PiCar** por lo que para ser utilizado en el coche real sin una fuente de alimentación, es necesario que esté conectado al coche **PiCar**. También cabe la posibilidad de conectar la placa **Raspberry Pi** a un ordenador portátil.

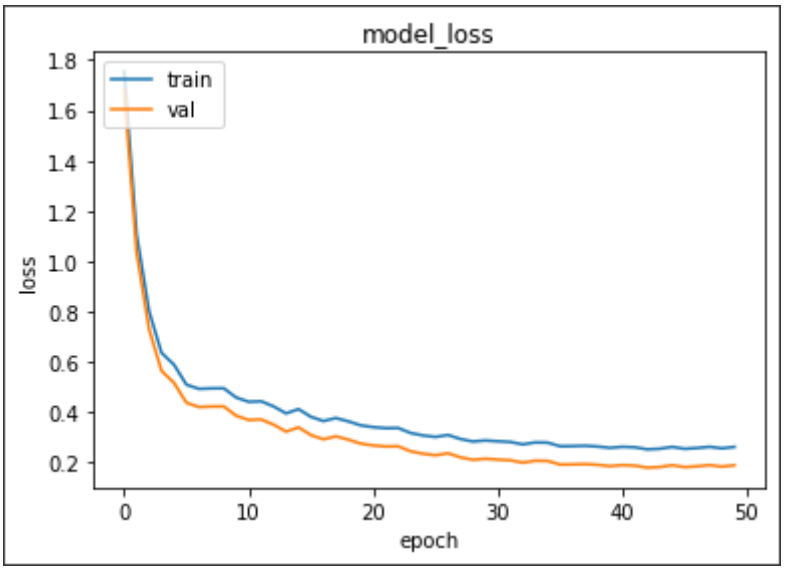


Figura 75: Gráfico de aprendizaje

Ahora que se dispone de un modelo que detecta señales de tráfico, se puede modificar el *script* anterior con la flecha indicando la dirección, para acomodar diferentes acciones en función de la señal detectada. Para este caso, se mostrará en la imágen detectada la velocidad actual a la que debería circular el coche en la parte superior izquierda, que irá modificándose si se detectan señales de límites de velocidad, si se detecta una señal de "STOP", la velocidad será de 0 y se mostrará la palabra "STOP" en la parte superior central durante tres segundos, y si se detecta una señal de paso de peatones, la velocidad se reduce a un valor de 20 y aparece la palabra "PRECAUCIÓN" en la parte superior central.

6. Resultados

Como resultado de este TFM, se ha conseguido que el coche **PiCar** reconozca las líneas del circuito por el cual circula y responde adecuadamente girando en la dirección correcta para mantenerse en el circuito (figura 76).

Figura 76: Resultado del coche PiCar recorriendo el circuito

A continuación que reconozca también las señales que tiene delante (figura 77) y actúe en consecuencia.

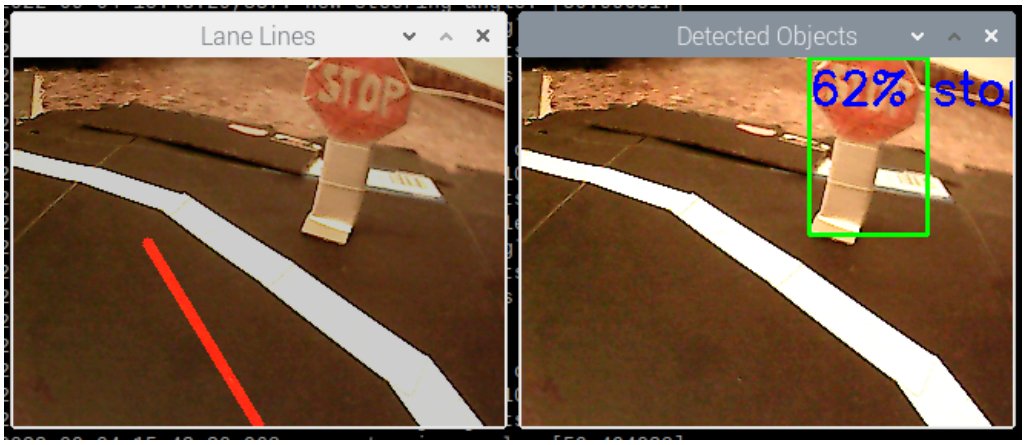


Figura 77: Reconocimiento de la señal “STOP”

Para adaptar lo realizado a una situación real y por tanto tener un caso práctico, se decide crear un sistema que actúe de coche autónomo ya que no se puede interactuar directamente con el mismo.
En esta parte se consigue crear un modelo que permite, por medio de la señal de una cámara, o si se requiere, de un vídeo, reconocer el escenario actual e indicar hacía dónde dirigirse a continuación (figura 78).

Figura 78: Línea indicando la dirección a seguir

Y que además detecte las diferentes señales y reaccione a ellas (figura 79).

Figura 79: Detección de señales

Al ser un trabajo académico, claramente no está destinado al uso en situaciones reales donde no se tenga control del vehículo. Lo redactado no tiene interacción con el coche, por lo que es más una ayuda al conducir.

La principal ventaja que se tiene utilizando una Raspberry Pi con Coral, es la posibilidad de tener un sistema de inteligencia artificial que cabe en el bolsillo, es fácil de acoplar cualquier tipo de dispositivo a este como puede ser una cámara, y realizar todo tipo de tareas.

A lo largo de este TFM se ha ido explicando cómo han ido cambiando muchas cosas en las tecnologías utilizadas en relación del artículo seguido, siendo esta otra desventaja, que como consecuencia de la evolución de las tecnologías, han habido problemas en la instalación, ejecución o funcionamiento.
Otra desventaja es la necesidad de tener una pantalla inicialmente para poder lanzar las tareas. Esto se puede remediar configurando las tareas en la **Raspberry Pi** para que se ejecuten una vez se inicie la placa, por ejemplo, con **Cron**[35].

7. Conclusiones

Al finalizar este proyecto, se puede afirmar que se han realizado satisfactoriamente todos los objetivos planteados inicialmente. Se ha conseguido tener un microordenador **Raspberry Pi** que realiza tareas con *Deep Learning* por medio del acelerador USB de **Coral**, siendo el coche **PiCar** el que detecta las señales y el acelerador USB realiza los cálculos de predicción. Además se ha conseguido adaptar el proceso seguido con el coche **PiCar** a una situación real, que tenga funcionalidades de un coche autónomo donde indica la dirección, velocidad y acciones a tomar cuando se detecte una señal de tráfico.
Todo esto se ofrece en este TFM a modo de guía para futuros lectores que deseen realizar *Deep Learning* en microordenadores, solucionando y mejorando los diferentes problemas que se pueden encontrar en otras guías y artículos más antiguos.
También se ha completado el objetivo enfocado a la fluidez entre el microordenador y el acelerador USB. El modelo resultante no excede los 8GB y no se transmite información por medio de un protocolo de cliente-servidor, resultando en latencia nula.

A nivel personal, he podido aprender más sobre la inteligencia artificial y cómo trabajar con una placa **Raspberry Pi**, que hasta ahora solamente había trabajado con placas Arduino. También he podido mejorar mis habilidades de programación en **Python**, y aprender a programar y configurar tareas de *Deep Learning* y lugares remotos y tamaño reducido, que será de utilidad para el futuro y en mi trabajo.

Todos los *scripts* y códigos de **Colab** utilizados se pueden encontrar en el repositorio de **Github**[48].

7.1 Relación con los estudios cursados

Para la realización de este TFM, se ha tenido en cuenta varias asignaturas cursadas durante el Máster que han sido de ayuda.
Empezando por la asignatura de “Sistemas inteligentes”, en la que se dieron nociones teóricas y técnicas y métodos de optimización sobre inteligencia artificial y autómatas. También se programó con **Python** varios trabajos. En la asignatura de “Computación de altas prestaciones” se enseñó a programar con GPUs y operaciones matriciales. La asignatura de “Sistemas empotrados y ubicuos” ofrecía conceptos teóricos sobre los diferentes sistemas empotrados y **Raspberry Pi**. La asignatura de “Ciencia de datos” es la que más está enfocada al tema de la inteligencia artificial, ya que se dieron nociones tanto teóricas como prácticas, de todo lo relacionado con la obtención y manipulación de datos. Siendo esto una gran parte de la creación de los modelos. Se utilizaron los lenguajes de programación **R** y **Python**.

7.2 Trabajos futuros

El proyecto realizado dispone de margen de mejora, tanto a la hora de entrenar un modelo, como de la obtención de datos.

La principal mejora que se puede aplicar es pasar el código de **Colab** a la nueva versión de **TensorFlow** y encontrar una forma de enlazar el modelo realizado en la versión 2 a la versión 1 de **Raspberry Pi**. Otra opción sería encontrar la forma de poder utilizar la versión 2 en la **Raspberry Pi** y de esta forma se puede entrenar el modelo en la misma versión.

Otra mejora que se puede aplicar si se plantea su uso en localizaciones remotas, es la posibilidad de incluir algún tipo de pantalla pequeña como se explicó en el apartado anterior ***6º Resultados***, o configurar **Cron**.

También es posible mejorar el modelo que se utiliza al recorrer la carretera con miles o incluso millones de imágenes. El modelo utilizado en el artículo de investigación realizado por Nvidia, se utilizan vídeos de carreteras con más de 70 horas de grabación, esto mejoraría mucho la precisión del modelo.
Para esto último sería de mucha utilidad tener un sistema EPS al cual se pueda conectar y obtener los diferentes datos que puede disponer el coche.

Anexo

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**		X		
ODS 7. **Energía asequible y no contaminante.**	X			
ODS 8. **Trabajo decente y crecimiento económico.**		X		
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.	X			
ODS 12. **Producción y consumo responsables.**		X		
ODS 13. **Acción por el clima.**	X			
ODS 14. **Vida submarina.**		X		
ODS 15. **Vida de ecosistemas terrestres.**		X		
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Para cumplir con los ODS marcados en la tabla anterior, el TFM presentado intenta conseguirlo con microordenadores, capaces de realizar tareas de inteligencia artificial, *deep learning* en este caso, para mejorar, tanto el funcionamiento, como el mantenimiento de los diferentes sectores que se puedan beneficiar de estas tareas.

Es posible monitorizar, detectar y actuar si se producen cambios en el agua, tanto visuales como microscópicos con los dispositivos adecuados. Si aparecen residuos sólidos o líquidos en el agua, se pueden detectar y eliminar. Además del agua, se puede realizar las mismas acciones con la estructura que contenga el agua o, en cuanto al saneamiento, por donde transita el agua. También es capaz de detectar imperfecciones en la estructura o funcionamiento incorrecto de algún aparato.

La energía utilizada por el microordenador es relativamente baja comparada a la utilizada por un ordenador de sobremesa u otro ordenador especializado. Existe también la posibilidad de acoplar baterías o dispositivos de energía renovable, como puede ser un panel solar.

Al ser microordenadores que se pueden dejar para que trabajen de forma independiente, no es necesario que usuarios, o trabajadores, se localicen cerca de estos una vez estén instalados, mejorando así, la forma de trabajar, especialmente en lugares inhóspitos. Al igual que ayuda al trabajador, puede ayudar a las empresas, tanto grandes como pequeñas, que utilicen estos sistemas para reducir costes y aumentar la producción, afectando a la empresa y al crecimiento económico.

En la industria su utilidad es amplia, desde control de calidad, obtención de datos, hasta control de robots. Es posible sustituir muchos de los sistemas actuales y maquinaria con los microordenadores con IA que afectaría a la sostenibilidad y la fomentación e innovación tecnológica. Además sería posible utilizar estos sistemas para controlar los residuos de las empresas.

Las ciudades y comunidades pueden utilizar estos dispositivos y las tareas de inteligencia artificial para controlar, ajustar, mantener y analizar infinidad de … Por ejemplo, detectar vehículos en varias calles para ajustar el tiempo de los semáforos y de esta forma reducir el tráfico y las emisiones de dióxido de carbono. Analizar el tráfico de ciertas partes de la ciudad o alrededores para sugerir modificaciones de la calzada, para eliminar o añadir carreteras. Ofrecer algún tipo de seguridad al detectar personal no autorizado.

Al igual que en la industria, en la producción es posible realizar control de calidad y automatización. Los microordenadores, al tener un consumo menor de electricidad, aportaría al consumo responsable de las empresas y usuarios.

Como se comentó en el punto anterior, los microordenadores utilizan muy poca energía, ayudando a reducir el consumo eléctrico procedente de combustibles fósiles. Además permite y facilita el uso de energías renovables, como la solar o eólica. Con las variedad de tareas que se pueden realizar, ayudaría a mejorar todo tipo de ecosistemas y climas. Al poder estar localizado en el mismo sitio, sin necesidad de desplazamientos o

transferencia de datos, se puede actuar en el instante al poder seleccionar la mejor solución posible.

También es posible utilizar microordenadores con *deep learning* para observar y controlar la vida submarina, o detectar objetos, estructuras o formaciones rocosas, de manera sencilla y rápida.

De manera similar a la vida submarina, es posible controlar, analizar y detectar fauna y flora terrestre. Con esto, se pueden crear modelos que ayuden, por ejemplo, en la sostenibilidad de animales en peligro de extinción, controlando el alimento y peligros que puedan tener entre otros factores. Detectar y contar flora en diferentes situaciones para lo que se requiera, como es el caso de agricultores, que pueden controlar si las verduras y frutas están maduras o si existen hierbas o insectos que dificulten el crecimiento de estás.

Bibliografía

[1] Uso de la inteligencia artificial - *thats ai*:
https://www.thats-ai.org/en-GB/units/ai-is-all-around-us

[2] Artículo de John McCarthy - *borghese*:
https://borghese.di.unimi.it/Teaching/AdvancedIntelligentSystems/Old/IntelligentSystems_2008_2009/Old/IntelligentSystems_2005_2006/Documents/Symbolic/04_McCarthy_whatisai.pdf

[3] Alan Turing - *wikipedia*: https://en.wikipedia.org/wiki/Alan_Turing

[4] Entscheidungsproblem - *science direct*:
https://www.sciencedirect.com/topics/mathematics/entscheidungsproblem

[5] Computing Machinery and Intelligence - *csee*:
https://www.csee.umbc.edu/courses/471/papers/turing.pdf

[6] Artículo sobre Google LaMDA - *endgadget*:
https://www.engadget.com/blake-lemoide-fired-google-lamda-sentient-001746197.html?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAAI8J6Pl8hNXPbeOIdy8aohO5SktHdojHzRbBwCEmqKT2admMBumm1nSUNCb9EdrwPgseR8R7tTJk0MDV-bcKd5hZBSZyai61QSlsyyYylHxFF0oTjNxcQ-MqnAEJK1dit1kjAtx1VjkC5xpydRqalf9Ee8-TkU5Sz01DfKeX1lj7

[7] - A logical calculus of the ideas immanent in nervous activity - *csulb*:
https://web.csulb.edu/~cwallis/382/readings/482/mccolloch.logical.calculus.ideas.1943.pdf

[8] Perceptron - *cite seer x*:
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf

[9] Paul Werbos, algoritmo de propagación - *research gate*:
https://www.researchgate.net/publication/35657389_Beyond_regression_new_tools_for_prediction_and_analysis_in_the_behavioral_sciences

[10] Andrew Ng y Jeff Dean - *ny times*:
https://www.nytimes.com/2012/06/26/technology/in-a-big-network-of-computers-evidence-of-machine-learning.html

[11] Deep learning - *ibm*: https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning

[12] Deep learning - *wikipedia*: https://en.wikipedia.org/wiki/Deep_learning

[13] Deep learning - *machine learning mastery*: https://machinelearningmastery.com/what-is-deep-learning/

[14] Artículo AplphaGo - *deep mind*: https://www.deepmind.com/research/highlighted-research/alphago/the-challenge-match

[15] LSTM - *arxiv*: https://arxiv.org/pdf/1503.04069.pdf

[16] Gated recurrent units - *arxiv*: https://arxiv.org/abs/1412.3555

[17] Red neuronal de células por Kunihiko Fukushima - *research gate*: https://www.researchgate.net/publication/222029751_Neocognitron_A_new_algorithm_for_pattern_recognition_tolerant_of_deformations_and_shifts_in_position

[18] Aplicación de reconocimiento facial utilizado en la Super Bowl en 2001 - *abc news*: https://abcnews.go.com/Technology/story?id=98871&page=1

[19] Coches autonomos - *nvidia*: https://www.nvidia.com/en-us/self-driving-cars/

[20] Uso de inteligencia artificial en médicina - *cancer*: https://www.cancer.gov/news-events/cancer-currents-blog/2022/artificial-intelligence-cancer-imaging

[21] Pinto model zoo - : https://github.com/PINTO0309/PINTO_model_zoo

[22] Keras: https://keras.io/

[23] TensorFlow: https://www.tensorflow.org/

[24] Caffe: http://caffe.berkeleyvision.org/

[25] PyTorch: https://pytorch.org/

[26] JAX: https://github.com/google/jax

[27] MXNet: https://mxnet.incubator.apache.org/versions/1.9.1/

[28] Quantización - *pytorch*: https://pytorch.org/docs/stable/quantization.html

[29] GPU - *wikipedia*: https://en.wikipedia.org/wiki/Graphics_processing_unit

[30] Edge computing - *wikipedia*: https://en.wikipedia.org/wiki/Edge_computing

[31] Edge computing - *red hat*:
https://www.redhat.com/en/topics/edge-computing/what-is-edge-computing

[32] Edge computing - *nvidia*:
https://blogs.nvidia.com/blog/2022/02/17/what-is-edge-ai/

[33] Edge device - *intel*:
https://www.intel.com/content/www/us/en/edge-computing/edge-devices.html

[34] Artículo sobre la escacez de chips - *bussiness insider*:
https://www.businessinsider.com/global-chip-shortage-could-last-another-2-years-experts-say-2021-5

[35] Mejoría de la escacez de chips - *reuters*:
https://www.reuters.com/technology/global-manufacturers-see-chip-shortage-easing-2022-07-21/

[36] Artículo sobre el coche PiCar - *towards data science*:
https://towardsdatascience.com/deeppicar-part-1-102e03c83f2c

[37] Pasos para preparar la Raspberry Pi - *raspberry pi*:
https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up

[38] Post con soluciones a posibles errores - *raspberry forum*:
https://forums.raspberrypi.com/viewtopic.php?t=58151

[39] Descarga del Imager - *raspberry pi*: https://www.raspberrypi.com/software/

[40] Imágenes para el sistema operativo - *raspberry pi*:
https://downloads.raspberrypi.org/raspbian/images

[41] RealVNC - *real vnc*: https://www.realvnc.com/en/connect/download/viewer/

[42] Documentación sobre la conectividad del coche PiCar - *sunfounder*:
https://docs.sunfounder.com/projects/picar-v/en/latest/circuits_building.html

[43] Instalación PyCoral - *coral*: https://coral.ai/docs/accelerator/get-started/

[44] Transformación de Hough - *opencv*:
https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html

[45] End to End Learning for Self-Driving Cars - *arxiv*: https://arxiv.org/abs/1604.07316

[46] LabelImg - *github*: https://github.com/tzutalin/labelImg

[47] Diferentes modelos a utilizar - *coral*: https://www.coral.ai/models/object-detection/

[48] Repositorio Github con todos los scripts utilizados - *github*: https://github.com/dagavUPV/TFMCodes

Glosario

(1) **Hardware**: Conjunto de elementos físicos o materiales que constituyen una computadora o un sistema informático.

(2) **TFM**: Trabajo Final de Máster

(3) **IoT**: *Internet of things*, o Internet de las cosas, dispositivo o grupo de dispositivos con sensores y *software* que se conectan y transmiten datos a través de Internet u otros sistemas de comunicación.

(4) **USB**: *Universal Serial Bus*, bus universal en serie. Bus de comunicación entre computadores, periféricos y dispositivos electrónicos.

(5) **Chatbot**: Es un programa dedicado a interactuar con una persona por medio de reconocimiento de voz e inteligencia artificial.

(6) **Programación lógica**: Es un paradigma de programación basado en lógica formal.

(7) **Reglas de producto**: Fórmulas matemáticas para hallar la derivada del producto de dos o más funciones.

(8) **Redes semánticas**: Representan relaciones semánticas entre conceptos en una red.

(9) **Frame**: una estructura de datos que divide los diferentes datos en subestructuras utilizado en inteligencia artificial.

(10) **Máquina Lisp**: es un ordenador de uso general diseñado específicamente para ejecutar programas escritos en Lisp. Lisp es un lenguaje de programación creado por John McCarthy con la finalidad de tener una notación matemática para programas de ordenador. Siendo el programa principal para los investigadores de la época de IA.

(11) **Etiquetas**: Proceso de desarrollo de un modelo donde se identifican objetos o imágenes y se les pone una etiqueta o nombre.

(12) **Go**: Juego de mesa de estrategia surgido en China hace más de 2500 años.

(13) **Sobreajuste**: Es la producción del análisis que corresponde exactamente con los datos aportados, fallando en predecir resultados al no poder amoldarse a nueva información.

(14) **3D**: tres dimensiones, representación geométrica de la altura, anchura y profundidad de un objeto.

(15) **RGB** (*Red, Green, Blue)* o Rojo, Verde, Azul: son los colores primarios de los cuales está formada cualquier imágen en el modelo RGB. Al juntarse con diferentes … se consiguen todos los colores en el espectro de colores. Existen otros modelos como RYB(*Red, Yellow, Blue*), CMY(*Cyan, Magenta, Yellow*) y CMYK(*Cyan, Magenta, Yellow, Key(black)*).

(16) **Gradiente descendiente**: Método iterativo de optimización de primer orden para encontrar el mínimo de una función.

(17) **Función de activación**: es la función que define cómo se transformará la suma de los pesos de la entrada en la salida del nodo o nodos en la capa de la red neural y si se debe de activar un nodo o no.

(18) **Open-source**: O código abierto, es el código de programas *software* que está disponible de forma gratuita.

(19) **Apache**: Servidor web de código abierto.

(20) **GPU**: *Graphics processing unit*, unidad de procesamiento gráfico. Coprocesador utilizado para el procesamiento gráfico o cálculo de operaciones en coma flotante.

(21) **TPU**: *Tensor Processing Unit*, unidad de procesamiento tensorial. Es un circuito integrado aplicación específica y aceleración de inteligencia artificial desarrollado por **Google** para ser utilizado en redes neuronales en *machine learning*.

(22) **CPU**: Circuito electrónico que interpreta y ejecuta instrucciones de programas.

(23) **Leaky ReLu**: Similar a la función ReLu con la diferencia

(24) **Bluetooth**: Tecnología de red inalámbrica de corto alcance que permite la transmisión de datos mediante radiofrecuencia.

(25) **Script**: En este TFM, se utiliza esta palabra para referirse a una serie de líneas de código, definidas en un fichero que, al ejecutarse, producen un resultado.

(26) **Github**: Sitio web "forja" utilizado principalmente para almacenar código fuente de programas.

(27) **CuDNN**: *NVIDIA CUDA Deep Neural Network*, es una biblioteca para aceleración de GPU diseñada por **Nvidia** para redes neuronales profundas.

(28) **Boot**: O *booting*, es el proceso de arrancar un ordenador.

(29) **Post**: Se refiere a un mensaje publicado típicamente en foros y blogs en Internet.

(30) **SSH**: *Secure Shell Protocol*, protocolo para realizar operaciones de red de forma segura, especialmente utilizado para acceso remoto.

(31) **API**: *Application programming interface*, interfaz de programación de aplicaciones. Interfaz que ofrece subrutinas, funciones y servicios a programas de ordenador.

(32) **Dataset**: Colección de datos.

(33) **DataFrame**: Estructura de datos tabulados.

(34) **Histograma**: Representación de la distribución de un conjunto de datos.

(35) **Cron**: Administrador de procesos en segundo plano en sistemas Unix.