



Universidad  
de Huelva



# Escuela Técnica Superior de Ingeniería

Grado en Ingeniería Informática

## Trabajo de Fin de Grado

---

Implementación de una red convolucional  
sobre un dispositivo móvil

# Resumen

El creciente uso de la inteligencia artificial aplicada a multitud de los actuales procesos industriales y automatización ha supuesto un reto a la hora de llevar esta tecnología a la mayoría de los lugares posibles, intentando, tanto abaratar el hardware usado como la velocidad y requisitos de este.

A lo largo de este documento explicaremos como aplicar técnicas de inteligencia artificial al campo de la visión artificial con la creación y posterior uso de las redes neuronales convolucionales sobre una placa MaixBit®, fabricada especialmente para albergar este tipo de tecnología en su interior.

Algo muy importante para tener en cuenta es el uso de software libre a lo largo de todo el proceso de desarrollo ya que todas las herramientas que vamos a usar son gratuitas y tienen, tanto una fácil accesibilidad como una gran potencia de desarrollo.

Con los resultados obtenidos como evidencia, queda demostrada la portabilidad de estos sistemas tan potentes como son las redes convolucionales a un hardware de tan poco coste y un tamaño reducido.

**Palabras clave:** inteligencia artificial, hardware, redes neuronales convolucionales, software libre, visión artificial, portabilidad.

---

## Abstract

Growing use of AI in many current industrial processes and automation has implied a challenge regarding portability and ease of use, in the fields of improvements of hardware and reduced costs.

In this document, Computer Vision AI techniques (specifically, convolutional neural networks) will be used on a MaixBit® board specially prepared to be used with this kind of technology.

The development process has mainly used open source software. This allows us to use powerful tools that are cost-free and publicly available for everyone.

This document aims to proof portability and effectiveness of these Computer Vision techniques in such a low-requirements system using several tests and results.

**Key words:** Artificial Intelligence, hardware, Convolutional Neural Networks, Open Source, Computer Vision, Portability.

---

# Índice General

<b>Capítulo 1. Propuesta del Proyecto .....</b>	- 1 -
1.1 Motivación .....	- 1 -
1.2 Objetivos .....	- 2 -
1.3 Hardware y Software.....	- 3 -
<b>Capítulo 2. Introducción .....</b>	- 5 -
2.1 Introducción al Deep Learning .....	- 5 -
2.2 Introducción a las placas Maix Bit.....	- 9 -
2.3 Historia del Aprendizaje Profundo .....	- 11 -
2.3.1 Etapa 1. Cibernetica (años 1940 – 1960) .....	- 11 -
2.3.2 Etapa 2. Conexionismo (años 1980 – 1995).....	- 12 -
2.3.2 Etapa 3. Deep Learning (años 2006 – actualidad).....	- 14 -
2.4 Estado Actual.....	- 15 -
<b>Capítulo 3. Fundamentos teóricos .....</b>	- 19 -
3.1 Redes Neuronales Convolucionales.....	- 19 -
3.2 Tipos de capas en las redes CNN .....	- 23 -
3.2.1 Capa de Convolución.....	- 23 -
3.2.2 Capa de Muestreo o Agrupación ( <i>Pooling</i> ) .....	- 26 -
3.2.3 Capa de Aplastamiento o <i>Flattening</i> .....	- 27 -
3.2.4 Capa de <i>Fully-Connected</i> .....	- 28 -
3.3 Introducción a las tecnologías usadas .....	- 29 -
3.3.1 Arquitectura RISC-V .....	- 29 -
3.3.2 Internet of Things (IoT) .....	- 31 -
3.3.3 TensorFlow Lite .....	- 33 -
3.3.4 MicroPython .....	- 34 -
3.3.5 Sistemas embebidos.....	- 35 -
3.4 Diseño de un modelo portátil .....	- 37 -
3.4.1 Limitaciones .....	- 38 -
3.4.2 Datos necesarios.....	- 38 -
3.4.3 Estructura de la red.....	- 41 -
3.4.4 Estudio de resultados .....	- 45 -
3.4.5 Migración a la placa .....	- 49 -

<b>Capítulo 4. Trabajo de experimentación .....</b>	<b>- 54 -</b>
4.1 Objetivos y materiales .....	- 54 -
4.2 Pasos previos a la implementación .....	- 57 -
4.3 Metodología .....	- 57 -
4.3.1 Generación del modelo.....	- 58 -
4.3.2 Estudio de resultados .....	- 67 -
4.3.3 Exportación del modelo a la placa .....	- 73 -
4.3.4 Ejecución del modelo .....	- 78 -
4.3.5 Ejecución automática del modelo.....	- 83 -
<b>Capítulo 5. Conclusiones y Trabajo Futuro .....</b>	<b>- 86 -</b>
<b>Capítulo 6. Bibliografía .....</b>	<b>- 89 -</b>
<b>Capítulo 7. Índice de figuras .....</b>	<b>- 94 -</b>
<b>Anexo I. Configuración de nuestro equipo.....</b>	<b>- 99 -</b>
<b>Anexo II. Pasos para la incorporación del modelo .....</b>	<b>- 101 -</b>

# Índice de Figuras

<b>Figura 1:</b> Detección de un perro con una placa de marca MaixBit® .....	- 2 -
<b>Figura 2.</b> Software utilizado para la implementación de la CNN.....	- 3 -
<b>Figura 3:</b> Diferentes etapas del Machine Learning .....	- 6 -
<b>Figura 4:</b> Esquema diferentes tipos de aprendizaje .....	- 7 -
<b>Figura 5:</b> Esquema diferentes tipos de problemas de Machine Learning .....	- 9 -
<b>Figura 6:</b> Resumen de características de la placa Maix Bit for Risc-V .....	- 10 -
<b>Figura 7:</b> Resumen de especificaciones Software y eléctricas de la placa.....	- 11 -
<b>Figura 8:</b> Estructura del Perceptrón simple .....	- 12 -
<b>Figura 9:</b> Funcionamiento del algoritmo Backpropagation sobre un perceptrón.....	- 13 -
<b>Figura 10:</b> Estructura de una red Deep Belief Networks.....	- 14 -
<b>Figura 11:</b> Ejemplo de reconocimiento de objetos en una imagen.....	- 15 -
<b>Figura 12:</b> Porcentaje de artículos sobre Machine Learning y Aprendizaje por Refuerzo ..	- 16 -
<b>Figura 13:</b> Coche percibiendo del entorno con el uso de sensores .....	- 17 -
<b>Figura 14:</b> Automatización de un personaje de un videojuego mediante redes neuronales ..	- 17 -
<b>Figura 15:</b> Reconocimiento de personas mediante técnicas de Machine Learning .....	- 18 -
<b>Figura 16:</b> Resumen y estructura algoritmo Backpropagation.....	- 20 -
<b>Figura 17:</b> Imágenes tratadas por diferentes capas de convolución.....	- 21 -
<b>Figura 18:</b> Estructura de una red neuronal convolucional.....	- 22 -
<b>Figura 19:</b> Red CNN con las características obtenidas en cada capa de forma esquemática .....	- 22 -
<b>Figura 20:</b> Ejemplo filtro emboss sobre una imagen.....	- 23 -
<b>Figura 21:</b> Fórmula matemática de la convolución .....	- 24 -
<b>Figura 22:</b> Salida de la capa de convolución .....	- 24 -
<b>Figura 23:</b> Capa de convolución + ReLU .....	- 25 -
<b>Figura 24:</b> Resumen ejemplificado de la etapa de convolución .....	- 26 -
<b>Figura 25:</b> Diferentes posibles imágenes de entrada.....	- 26 -
<b>Figura 26:</b> Ejemplo Max Pooling 2x2 .....	- 27 -
<b>Figura 27:</b> Ejemplo aplastamiento de una matriz .....	- 28 -
<b>Figura 28:</b> Ejemplo de clasificación de una imagen de un perro.....	- 28 -
<b>Figura 29:</b> Estructura del procesador Kendrite K210.....	- 30 -
<b>Figura 30:</b> Uno de los procesadores RISC-V por la marca SiFive .....	- 31 -
<b>Figura 31:</b> Internet of Things .....	- 32 -
<b>Figura 32:</b> Logo TensorFlow Lite.....	- 33 -
<b>Figura 33:</b> MicroPython .....	- 34 -
<b>Figura 34:</b> Componentes de los sistemas embebidos .....	- 36 -
<b>Figura 35:</b> Sistemas embebidos presentes en un coche .....	- 37 -
<b>Figura 36.</b> Diferencia entre los tipos de padding .....	- 38 -
<b>Figura 37.</b> Imágenes etiquetadas del dataset.....	- 39 -
<b>Figura 38.</b> Ejemplo de Data Augmentation .....	- 40 -
<b>Figura 39.</b> Estructura de Red VGG 16 .....	- 42 -
<b>Figura 40.</b> Descenso por gradiente.....	- 43 -
<b>Figura 41.</b> Diferencia entre los diferentes descensos por gradiente .....	- 44 -
<b>Figura 42.</b> Algoritmo de optimización Adam.....	- 44 -
<b>Figura 43.</b> Comparación algoritmos de optimización (coste entrenamiento) .....	- 45 -
<b>Figura 44.</b> Datos obtenidos en cada una de las etapas .....	- 45 -

<b>Figura 45.</b> Ejemplo de Matriz de Confusión .....	- 46 -
<b>Figura 46.</b> Valores obtenibles a partir de la matriz de confusión .....	- 47 -
<b>Figura 47.</b> Diferencia entre sensibilidad alta o baja .....	- 48 -
<b>Figura 48.</b> Diferencia entre especificidad alta y baja.....	- 48 -
<b>Figura 49.</b> Implementación en Dispositivos de TFLite.....	- 50 -
<b>Figura 50.</b> Interfaz de la aplicación K-Flash .....	- 51 -
<b>Figura 51.</b> MaixPy IDE.....	- 52 -
<b>Figura 52.</b> Interfaz Putty.....	- 53 -
<b>Figura 53.</b> Placa Maix Bit.....	- 55 -
<b>Figura 54.</b> Sensor óptico OV2640 ojo de pez 2 Mpx .....	- 55 -
<b>Figura 55.</b> Pantalla LCD .....	- 56 -
<b>Figura 56.</b> Cables y Leds .....	- 56 -
<b>Figura 57.</b> Primera estructura generada .....	- 58 -
<b>Figura 58.</b> Estructura común a nuestros modelos .....	- 59 -
<b>Figura 59.</b> Gráfica de perdida sobre el modelo base.....	- 60 -
<b>Figura 60.</b> Gráfica de precisión sobre el modelo base .....	- 60 -
<b>Figura 61.</b> Funcionamiento de la capa de GAP.....	- 61 -
<b>Figura 62.</b> Data Augmentation aplicado en nuestro modelo. ....	- 61 -
<b>Figura 63.</b> Estructura de red reducida .....	- 62 -
<b>Figura 64.</b> Gráfica de pérdida del modelo reducido (Aprox. <b>0,14</b> min).....	- 63 -
<b>Figura 65.</b> Gráfica de precisión del modelo reducido (Aprox. <b>0,95</b> max).....	- 63 -
<b>Figura 66.</b> Funcionamiento del clasificador SoftMax .....	- 64 -
<b>Figura 67.</b> Estructura del modelo final .....	- 65 -
<b>Figura 68.</b> Gráfica de pérdida en el modelo final (Aprox. <b>0,15</b> min) .....	- 66 -
<b>Figura 69.</b> Gráfica de precisión en el modelo final (Aprox. <b>0,95</b> max).....	- 66 -
<b>Figura 70.</b> Matriz de confusión generada.....	- 67 -
<b>Figura 71.</b> Imagen de gato predicha correctamente con porcentaje del <b>71%</b> .....	- 68 -
<b>Figura 72.</b> Imagen de perro predicha correctamente con porcentaje del <b>98%</b> .....	- 68 -
<b>Figura 73.</b> Imagen de gato predicha correctamente con porcentaje del <b>99,9%</b> .....	- 69 -
<b>Figura 74.</b> Imagen de perro clasificada incorrectamente con un porcentaje del <b>78%</b> .....	- 69 -
<b>Figura 75.</b> Imagen de gato clasificada incorrectamente con porcentaje del <b>75,9%</b> .....	- 70 -
<b>Figura 76.</b> Tabla con datos de comportamiento de la red.....	- 70 -
<b>Figura 77.</b> Imagen de león pasada por el modelo.....	- 71 -
<b>Figura 78.</b> Imágenes de clases desconocidas .....	- 72 -
<b>Figura 79.</b> Estructura de nuestro entorno de trabajo .....	- 73 -
<b>Figura 80.</b> Ventana de comandos de Ubuntu Win10 .....	- 74 -
<b>Figura 81.</b> Transformación a kmodel desde el terminal de Ubuntu .....	- 75 -
<b>Figura 82.</b> Interfaz K-Flash GUI configurada .....	- 76 -
<b>Figura 83.</b> Proceso de carga en la placa .....	- 76 -
<b>Figura 84.</b> Placa con firmware cargado.....	- 77 -
<b>Figura 85.</b> Configuración para cargar el modelo a la placa.....	- 78 -
<b>Figura 86.</b> Imports utilizados en el script de MicroPython.....	- 79 -
<b>Figura 87.</b> Inicialización de los pines de salida.....	- 79 -
<b>Figura 88.</b> Esquema de la placa con los leds .....	- 80 -
<b>Figura 89.</b> Función para el manejo de los leds .....	- 80 -
<b>Figura 90.</b> Inicialización del sensor .....	- 81 -
<b>Figura 91.</b> Inicialización de nuestro LCD .....	- 81 -
<b>Figura 92.</b> Creación de etiquetas y carga del modelo.....	- 81 -

<b>Figura 93.</b> Clasificación de la imagen con nuestro modelo .....	- 82 -
<b>Figura 94.</b> Resultados finales de la ejecución .....	- 82 -
<b>Figura 95.</b> Interfaz inicial de la placa desde Putty .....	- 83 -
<b>Figura 96.</b> Interfaz principal de la ampliación uPyLoader.....	- 84 -
<b>Figura 97.</b> Interfaz con el dispositivo conectado.....	- 84 -
<b>Figura 98.</b> Editor de textos de uPyLoader.....	- 85 -
<b>Figura 99.</b> Modelo final de la placa. Vista Superior .....	- 88 -
<b>Figura 100.</b> Modelo final de la placa. Vista Inferior .....	- 88 -

---

# Capítulo 1

## Propuesta del Proyecto

---

En este capítulo introductorio desarrollaremos las diferentes motivaciones que nos han llevado a desarrollar este proyecto junto con los principales objetivos de los que partimos y las diferentes tecnologías y herramientas que han hecho falta para el desarrollo y la implementación.

### 1.1 Motivación

---

A lo largo de los años hemos visto como la tecnología ha ido introduciéndose en nuestras vidas hasta el tal punto de depender de ellas para casi todos los factores de nuestro día a día. Durante la historia han ido sucediendo grandes avances que han permitido desarrollar dispositivos cada vez más pequeños capaces de incluso duplicar la capacidad de computo de sus antecesores de no mucho más de 10 años, desecharando muchas de las tecnologías que han ido surgiendo con pocos años de uso.

Actualmente muchos expertos hablan de la segunda revolución industrial, donde la implementación de las nuevas tecnologías en las diferentes compañías ha supuesto un cambio radical en la forma de trabajo, aumentando la productividad y la calidad de los productos vendidos.

La informática es uno de los principales recursos en los que se apoyan estas nuevas tecnologías, siendo necesaria para la programación de los nuevos dispositivos y la incorporación de nuevos algoritmos capaces de optimizar las diferentes actividades a realizar.

Los algoritmos relacionados con la Inteligencia Artificial son los que han supuesto un mayor cambio sobre las operaciones que se podían realizar por computación y cuáles no, haciendo posible lo que hace años parecía imposible.

En nuestro caso, el principal recurso de esta rama es el referente a la visión artificial, la cual es capaz de percibir el mundo, tomar decisiones respecto a lo recibido en la etapa anterior y actuar según un criterio propio. En definitiva, son capaces de actuar como un humano, algo que era impensable unos años atrás.

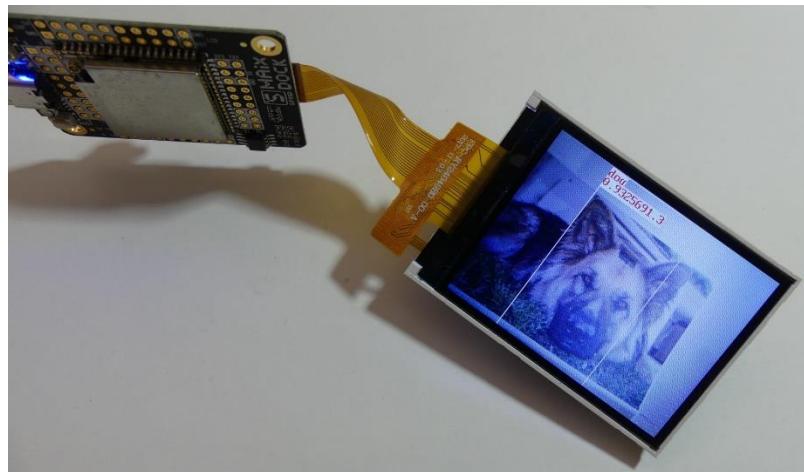
El principal motivo de este Trabajo de Fin de Grado es comprender el funcionamiento de esta herramienta tan poderosa como es el aprendizaje profundo, o *Deep Learning* en inglés, el cual ha sido el principal precursor de la inteligencia artificial tal y como la conocemos en la actualidad, ofreciéndonos unos resultados impensables para el mundo digital y que ahora son posibles de implementar de forma más o menos sencilla en un tiempo no muy extenso.

El continuo avance de estas tecnologías nos plantea nuevos problemas de movilidad y computabilidad, limitados hasta ahora al uso de un gran computador capaz de recoger estos algoritmos y ejecutarlos.

Actualmente no es difícil encontrar diferentes documentos dirigidos a la implementación de las nuevas tecnologías en dispositivos cada vez más pequeños, reducidos tanto en capacidad como en velocidad, suponiendo un reto para las empresas.

Un ejemplo de esto pueden ser las nuevas implementaciones que se están creando para reducir el volumen de un modelo intentando mantener los resultados que genera uno con mayor capacidad [[Xiangyu Zhang et al, 2018](#)].

Sumándonos a la solución de nuevos retos, nuestra propuesta es llevar un algoritmo clasificador a una pequeña placa, la cual cuenta con todas las herramientas necesarias para su ejecución, además de sus pequeñas dimensiones y bajo consumo, siendo una opción perfecta para su uso en el trabajo del futuro.



**Figura 1:** Detección de un perro con una placa de marca MaixBit®

## 1.2 Objetivos

Para la realización de este trabajo hemos de desarrollar dos principales objetivos a llevar a cabo:

- El primero de los dos objetivos generales del proyecto sería elaborar una pequeña red neuronal convolucional capaz de hacer una clasificación binaria sobre un sistema simple con la misión de familiarizarnos con este tipo de trabajo y entorno para ayudarnos a realizar posibles tareas futuras que usen la misma tecnología de una forma más rápida y sencilla.
- El segundo correspondería con la implementación del modelo generado en primera instancia en la placa mencionada en la introducción, generando un submodelo de menores requerimientos y exportándolo de forma que sea capaz de ejecutarse únicamente con el uso de la placa.

Estos dos grandes objetivos requieren de otros subobjetivos específicos que son con los que se ha desarrollado el TFG.

- Estudio de los conceptos básicos relacionados con el Deep Learning y la visión artificial.
- Estudio del funcionamiento de las placas MaixBit® y sus posibles usos.
- Aprender las diferentes capas que componen a las redes neuronales convolucionales (CNN) junto a las características de estas.
- Comprensión y definición de las tecnologías usadas por los diferentes dispositivos usados a lo largo de la elaboración del proyecto.
- Aprender como elaborar un modelo de visión artificial completo junto a su posterior estudio de resultados.
- Implementación y transformación de este modelo a uno más sencillo capaz de introducirse en la placa.
- Estudio de los diferentes softwares pertenecientes a la placa necesarios para desarrollar su funcionamiento.
- Completar una guía sobre el uso de este dispositivo accesible para cualquier posible empresa que requiera de esta tecnología en su trabajo.

### 1.3 Hardware y Software

Para poder llevar a cabo los objetivos mencionados anteriormente de una forma correcta es necesario el uso de un equipo hardware y software especializado en la aplicación de los algoritmos de visión por computador y los programas necesarios para el funcionamiento de la placa.

La implementación de la red neuronal convolucional la vamos a hacer en código Python 3.7 sobre la aplicación Anaconda que incorpora una instalación rápida del framework Tensorflow 2.0, que incluye en su interior a su vez Keras, una herramienta que nos facilita el uso y creación de los modelos CNN.

Set up Anaconda, Jupyter Notebook  
Install TensorFlow and Keras  
for studying Deep Learning



*Figura 2. Software utilizado para la implementación de la CNN*

La elección de este software ha sido debida a sus numerosas referencias en todo el mundo del *Deep Learning* junto a la multitud de ejemplos sobre los que se usan estas herramientas libres y que tantos recursos nos permiten usar. Estas herramientas las usaremos sobre un ordenador portátil capaz de procesar una gran cantidad de información y de procesar una nube inmensa de datos en poco tiempo usando una de las gráficas punteras en el mercado.

Las características de este ordenador estarán expuestas en el [Anexo I](#), donde explicaremos los componentes que lo contienen y la funcionalidad de cada uno de ellos en el ámbito de nuestro problema.

En la segunda parte del trabajo necesitaremos los recursos necesarios para conectar nuestra placa al ordenador, de forma que podamos cargarle los datos de nuestro modelo y podamos programarla según el trabajo que necesitamos realizar.

Como la mayoría de los posibles entornos donde pudiera realizarse la gestión de la placa son máquinas con Windows, hemos decidido trabajar sobre las aplicaciones que el propio fabricante tiene disponible, y que son de gran utilidad tanto para flashear la placa ([K-Flash GUI](#)) como para trabajar de forma sencilla en el entorno de MicroPython ([MaixBit IDE](#)).

Otras de las herramientas que necesitamos son las que nos proporciona la propia marca para transformar el modelo generado en Python con TensorFlow (model) en uno legible para la placa, que trabaja con subtipo específico de TensorFlow Lite, llamado '*kmodel*'. Toda esta gestión se realiza con los datos obtenidos del siguiente [repositorio](#) en GitHub.

Dado a la especificidad de los programas y a la dificultad de uso de estos he decidido generar un [Anexo II](#), donde explicaré paso a paso como conseguir que la placa se conecte y funcione correctamente con los modelos generados por nosotros.

---

# Capítulo 2

## Introducción

---

Debido a que el TFG está dividido en dos partes fundamentales, vamos a plantear el desarrollo de la memoria en estas dos mismas.

En la primera haremos una pequeña introducción sobre el aprendizaje automático que especificaremos en la primera parte del trabajo con el uso de las redes neuronales convolucionales.

Posteriormente haremos una pequeña mención a la placa que vamos a usar junto a algunas de sus características que hacen que su incorporación en este proyecto sea tan interesante.

Por último, hablaremos sobre los antecedentes de esta tecnología que, aunque haya alcanzado su auge hace pocos años, lleva desarrollándose y estudiándose mucho tiempo hasta llegar al estado actual de esta tecnología.

### 2.1 Introducción al Deep Learning

---

El aprendizaje automático es una de las tecnologías más rompedoras en la actualidad debido a su gran impacto en la sociedad y a su gran efectividad en los principales problemas. Este tipo de algoritmos ha ofrecido soluciones a problemas imposibles de afrontar con el uso de otras tecnologías.

El concepto de “aprendizaje automático” nos lleva a pensar en cómo se puede aplicar el concepto de la autonomía de una estructura o ente para “auto aprender” de lo que conoce del problema.

Visto de una forma más simple, es increíble como una estructura inicialmente sin conocimiento acaba desarrollando una solución a partir de los conceptos que “ve” y de la respuesta asociadas a estos.

Este fenómeno se produce en el mundo biológico constantemente ya que, por ejemplo, nosotros los humanos no dejamos de aprender gracias a los sensores y las respuestas que nos producen al realizar cierta acción. Por ejemplo: Si acerco la mano al fuego me quemo, por lo tanto, evitaré acercarme al fuego en el futuro.

Acercándonos al ámbito de la informática, nos damos cuenta de que, para resolver cualquier tipo de problema, debíamos elaborar un algoritmo capaz de estudiar todos los datos de entrada para desarrollar una salida dependiente del problema en cuestión.

El aprendizaje automático surge cuando nos encontramos problemas en los que no somos capaces de desarrollar un algoritmo trivial para llegar a su solución, como son los relacionados con la visión artificial, el reconocimiento del habla o la conducción autónoma.

Para solucionar este tipo de problemas nada triviales generamos una nueva forma de programar los diferentes algoritmos que conocíamos hasta el momento, usando los conceptos mencionados al inicio de esta sección.

Introducimos el concepto de aprendizaje en el ámbito de la informática, haciendo que el propio desarrollo del algoritmo se base en el conjunto de datos de entrada, elaborando un modelo que busque similitudes o patrones en este sin la necesidad de indicar explícitamente los pasos a seguir.

Este tipo de algoritmos son los que se denominan *Machine Learning* y se divide principalmente en dos tipos de etapas: la primera se dedica a la preparación de los datos de entrada para generar un modelo de la forma más efectiva posible y, en la segunda, usamos ese modelo generado para reconocer nuevos datos y elaborar resultados.

## The Machine Learning Process



**Figura 3:** Diferentes etapas del Machine Learning

Como podemos ver en la Figura 3 las etapas de estudio del modelo, elaboración de gráficos y tablas de precisión son muy importantes para comprender el funcionamiento del modelo y poder darlo como válido.

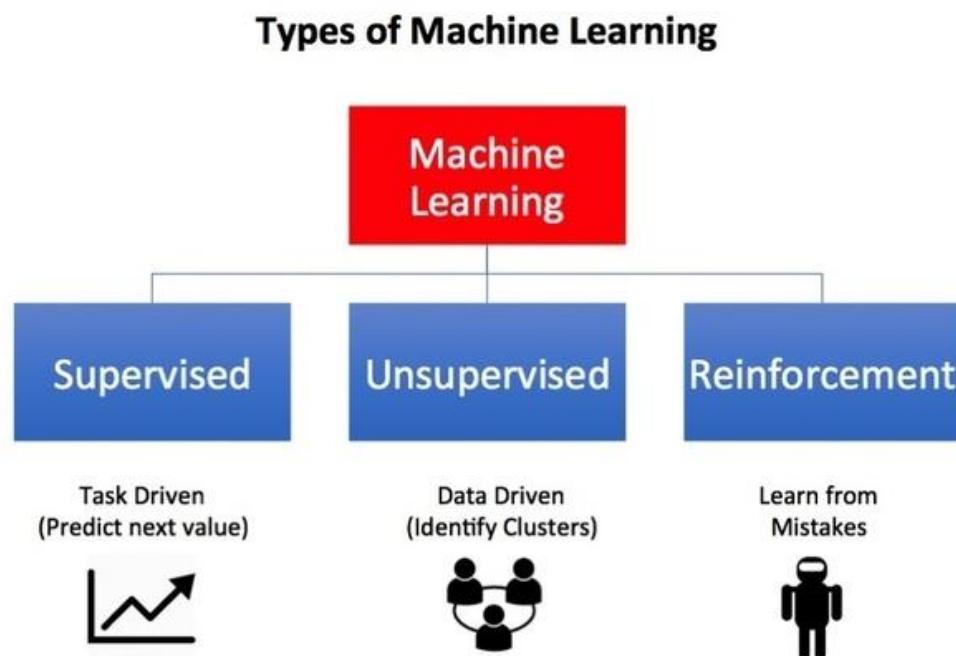
Estos modelos, al generarse de forma automática sobre una estructura preestablecida por el ingeniero, no pueden estudiarse en profundidad, por ello, los datos resultantes toman tanta importancia en estos algoritmos.

Dentro de la disciplina del *Deep Learning* podemos diferenciar diferentes algoritmos que son usados para diferentes propósitos dependiendo del problema que enfrenten. Podemos diferenciar tres tipos de aprendizaje:

- **Aprendizaje Supervisado:** En este tipo de aprendizaje conocemos en todo momento la salida para un conjunto de entrada determinado. Generalmente el modelo intenta conseguir que los datos de entrada lleguen a dar los resultados conocidos.
- **Aprendizaje no Supervisado:** No conocemos la salida para el conjunto de entrada determinado. Generalmente el modelo busca similitudes dentro de los datos de entrada.

- **Aprendizaje por refuerzo:** No representamos el comportamiento deseado mediante ejemplos, sino mediante una cierta evaluación sobre los resultados que genera el sistema en su entorno. El modelo se genera con la función recompensa/castigo.

En la figura 4 podemos ver un pequeño esquema de los diferentes tipos de aprendizaje.



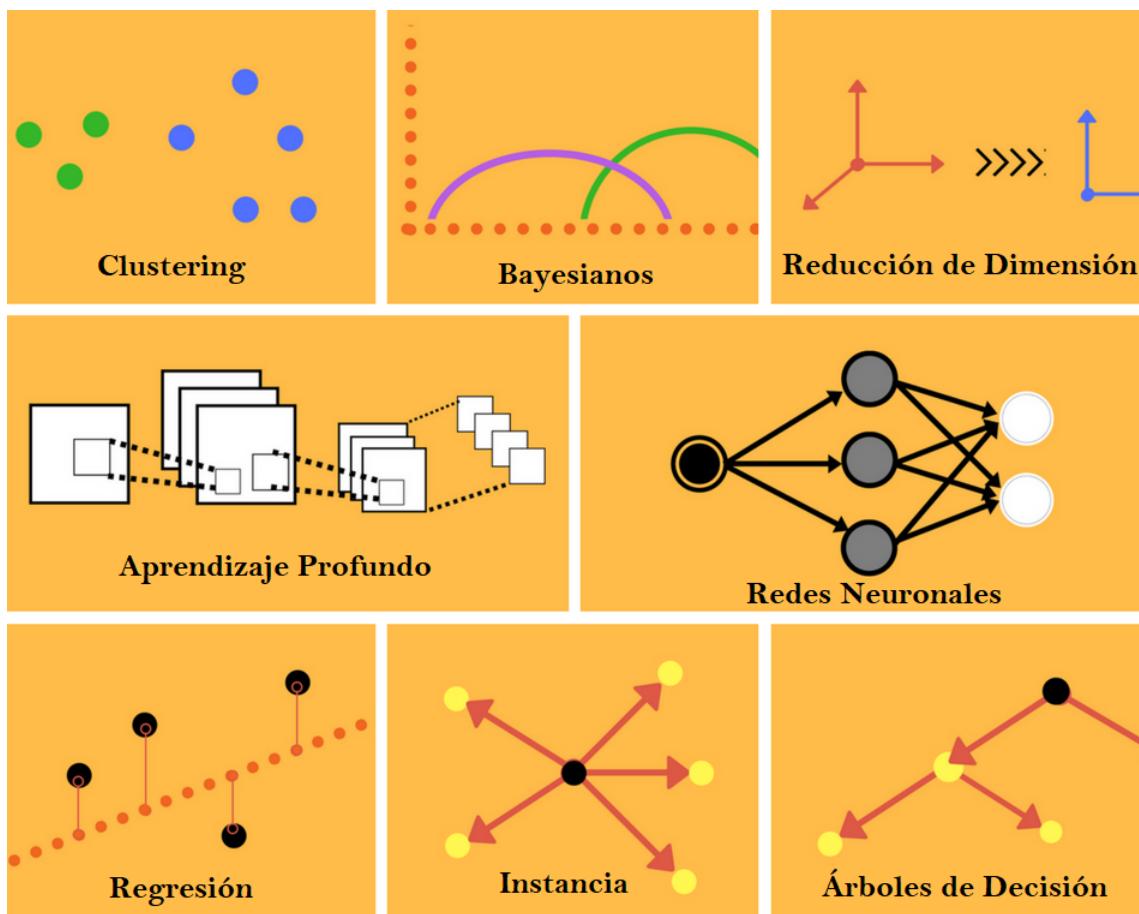
*Figura 4: Esquema diferentes tipos de aprendizaje*

Una vez vistos los diferentes tipos de problemas dependiendo de la forma de elaborar el modelo, podemos elaborar una clasificación de los diferentes aprendizajes en cuestión del conocimiento de salida que podemos obtener. Podemos diferenciar tres tipos:

- **Aprendizaje simbólico:** Trata de adquirir conocimiento que permita representaciones de conceptos (reglas lógicas, valores de atributos...).
- **Aprendizaje numérico:** Trata de adquirir un conocimiento representado por medio de factores numéricos no relacionados directamente con un concepto determinado (pesos de una red neuronal...)
- **Aprendizaje mixto:** Trata de adquirir conceptos expresados por medio de factores numéricos. Relaciones Valor-Número.

Para finalizar la introducción al Machine Learning vamos a hablar de diferentes problemas que se están solucionando con estas técnicas y algunos de sus principales algoritmos por cada uno de ellos:

- **Algoritmos de Regresión:** Nos ayudan a clasificar o predecir valores. Se intentará compensar la mejor respuesta a partir del menor error. Ejemplos de algoritmos: Regresión Lineal, Regresión Logística.
- **Algoritmos basados en Instancia:** Se usan para problemas de decisión con ejemplos de datos de entrenamiento que son importantes o requeridos por el modelo. Ejemplo de algoritmo: K-vecinos más cercanos (kNN)
- **Algoritmos de Árbol de Decisión:** Se usa como clasificador de información bifurcando los posibles caminos por probabilidad de ocurrencia. Intentará buscar el mejor árbol balanceado para clasificar el resultado. Ejemplos de algoritmos: Árboles de Clasificación y Regresión (CART), Árbol condicional.
- **Algoritmos Bayesianos:** Utilizan explícitamente el teorema de Bayes de probabilidad para problemas de clasificación y regresión. Ejemplos de algoritmos: *Naive Bayes*, *Bayesian Network*.
- **Algoritmos de Clustering (agrupación):** Se usa en entornos de aprendizaje no supervisado y consiste en agrupar los datos existentes de los que desconocemos sus características en común. La forma de agruparlos se basa en la cercanía. Ejemplos de algoritmos: *K-means*, *Hierarchical Clustering*.
- **Algoritmos de Redes Neuronales:** Son los algoritmos y estructuras inspirados en las funciones biológicas de las redes neuronales. Se usan tanto para problemas de clasificación como para problemas de regresión, aunque tienen un gran potencial para resolver cualquier tipo de problema. Son la base del aprendizaje profundo que es el que usaremos en este trabajo y estuvieron limitadas por la gran necesidad de procesamiento. Ejemplos de algoritmos: Perceptron, *Back-Propagation*.
- **Algoritmos de Aprendizaje Profundo:** Son la evolución de las Redes Neuronales Artificiales que aprovechan el gran avance de la tecnología y el abaratamiento de esta. Consisten en explotar una gran cantidad de datos en enormes redes neuronales interconectadas en diversas capas que pueden ejecutar en paralelo. Ejemplos de algoritmos: Convolutional Neuronal Networks (CNN), Long Short Term Memory Neuronal Networks (LSTNN).
- **Algoritmos de Reducción de Dimensión:** Buscan explotar la estructura existente de manera no supervisada para simplificar los datos y reducirlos o comprimirlos. Ejemplo de algoritmo: Principal Component Analysis (PCA).
- **Procesamiento del Lenguaje Natural:** Tiene como objetivo comprender el lenguaje humano, tanto de forma escrita como de forma oral. Tienen multitud de usos, desde analizar sintácticamente hasta el análisis de sentimientos en redes sociales. Este tipo de algoritmos son lo que estamos viendo cada vez más en nuestras vidas con la llegada de los asistentes virtuales como Siri o Cortana.



**Figura 5:** Esquema diferentes tipos de problemas de Machine Learning

## 2.2 Introducción a las placas Maix Bit

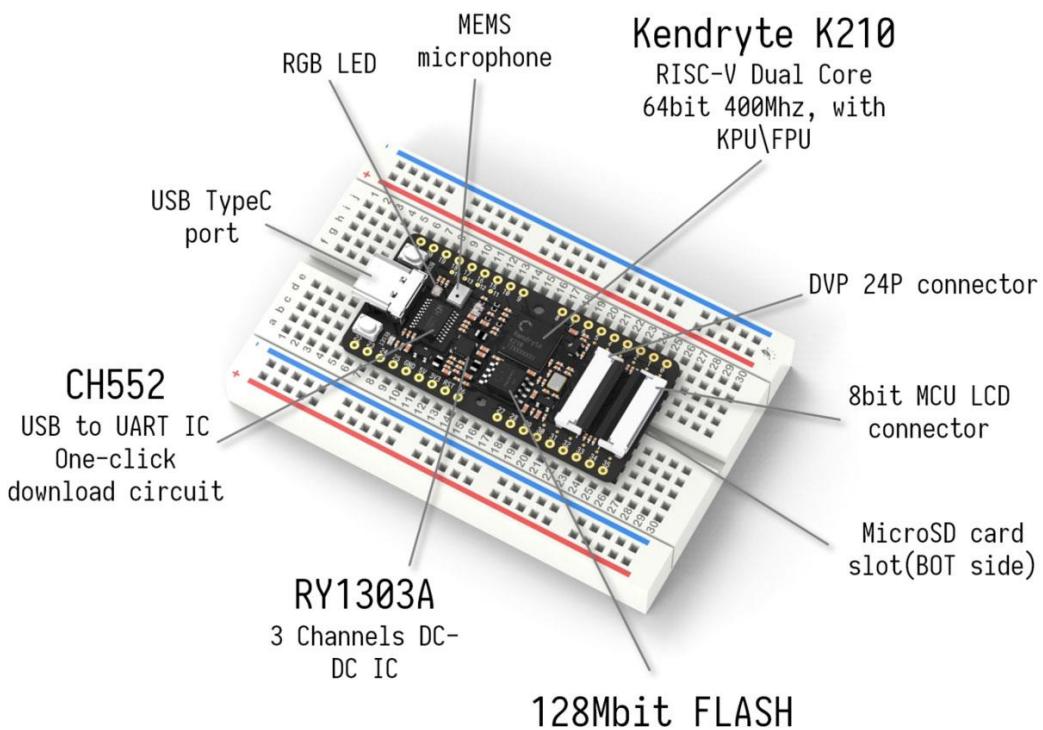
---

Las placas Maix Bit pertenecen a uno de los modelos que tienen disponibles la marca [Seeed](#) como dispositivos portátiles para la incorporación de diferentes tecnologías.

Esta marca se dedica a fabricar multitud de dispositivos electrónicos para desarrollar multitud de tareas de diferente índole, en nuestro caso, nos interesa los dispositivos dedicados al ámbito de la inteligencia artificial, más concretamente en el aprendizaje profundo con el uso de las redes neuronales convolucionales (CNN).

La placa que vamos a usar durante el desarrollo del trabajo es la denominada [Sipeed MAix BiT for RISC-V AI+IoT](#) la cual está especialmente diseñada para controlar arquitecturas de hardware libre como la [Risc-V](#) y elementos de inteligencia artificial como las redes convolucionales que implementaremos en la primera parte de la práctica.

Una de las características más visuales de la placa son las medidas tan pequeñas que implementa y es que tan solo mide 5,3 centímetros de ancho y poco más de 2,5 cm de largo, convirtiéndola en un dispositivo de gran portabilidad que de manera sencilla se puede implementar en cualquier entorno de trabajo, que es el objetivo en definitiva de este trabajo.



**Figura 6:** Resumen de características de la placa Maix Bit for Risc-V

Una de las características que más llama la atención de esta placa es el potente procesador que incluye en tan pequeñas dimensiones, como es el Kendrite K210 que es un procesador doble núcleo con la tecnología Risc-V de 64 bits, el cual alcanza velocidades de hasta 800 MHz usando *overclocking*.

Todas estas características convierten a la placa en una herramienta perfecta para incorporar diferentes modelos generados con técnicas basadas en *Deep Learning* de forma efectiva y trabajar con ellas en cualquier tipo de entorno.

Esta placa trabaja en un entorno controlado de [MicroPython](#), un lenguaje de programación de poco peso y funciones sencillas que controla las diferentes funciones de la placa.

Para el funcionamiento del modelo CNN en el interior de la placa deberemos correr el modelo con el uso del [TensorFlow Lite](#), una versión reducida del framework que usaremos para entrenar el modelo de clasificación binaria.

Por último, es de gran importancia mencionar que este dispositivo se trata de un sistema embebido que funciona a tiempo real, es decir, es capaz de realizar tareas muy específicas únicamente con los datos de entrada generados en un instante y generando una salida acorde a ellos.

En la figura 7 se incluye un pequeño repaso general a las diferentes características que incluye la placa, así como el consumo de energía y los diferentes rangos de temperatura en los que puede trabajar.

SOFTWARE FEATURES	
FreeRtos & Standard SDK	Support FreeRtos and Standrad development kit.
MicroPython Support	Support MicroPython on M1
Machine vision	Machine vision based on convolutional neural network
Machine hearing	High performance microphone array processor

ELECTRICAL SPEC	
Supply voltage of external power supply	4.8V ~ 5.2V
Supply current of external power supply	>600mA
Temperature rise	<30K
Range of working temperature	-30°C ~ 85°C

**Figura 7:** Resumen de especificaciones Software y eléctricas de la placa

## 2.3 Historia del Aprendizaje Profundo

---

Como hemos mencionado anteriormente, el aprendizaje profundo no es más que una de las estrategias que han surgido a partir de los sistemas biológicos como las redes neuronales llevadas a ámbitos más específicos.

Este concepto nos lleva a pensar que hablar de la historia del *Deep Learning* es hablar también de la historia del *Machine Learning*, mencionando los principales hitos que han ido sucediendo a lo largo de los años hasta llegar al conocimiento que actualmente tenemos sobre este terreno.

La historia del Machine Learning se puede dividir en tres etapas fundamentales en las que se pueden mencionar algunos de los logros más exitosos y de las estructuras más famosas que aún se siguen usando.

### 2.3.1 Etapa 1. Cibernetica (años 1940 – 1960)

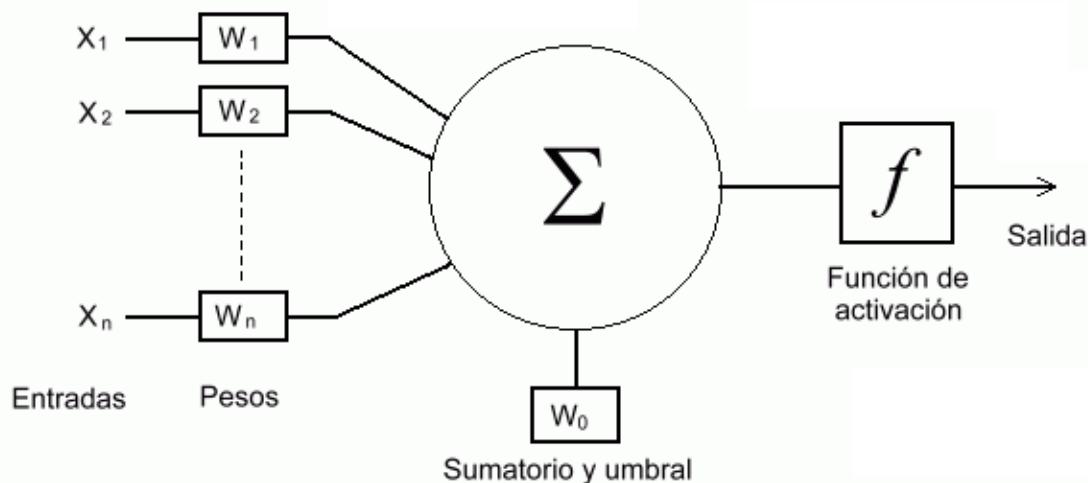
---

Esta primera etapa arrancó con los estudios sobre el aprendizaje biológico recogido por primera vez en la documentación de [\[McCulloch&Pitts, 1943\]](#) y [\[Hebb, 1949\]](#) que a su vez dieron pie a las primeras implementaciones informáticas apoyadas por estos estudios como el modelo de Perceptrón presentado por Rosenblatt en 1958 [\[Rosenblatt, 1958\]](#).

En esta primera etapa el principal hito que se dio y que supuso el inicio de las redes neuronales artificiales fue la creación del perceptrón por Rosenblatt inspirado en el trabajo de Warren McCulloch y Walter Pitts.

Esta estructura toma varias entradas binarias y produce una única salida, la cual se calcula con la multiplicación de los valores de entrada por los pesos introducidos por su autor que posteriormente se sumarían y se le aplicaría una función de activación que haría que se activase o no la salida en función de un umbral.

Las principales operaciones que podía implementar se resumían en decisiones binarias sencillas o la creación de funciones lógicas como las puertas lógicas OR o AND.



**Figura 8: Estructura del Perceptrón simple**

Poco después del perceptrón se creó la primera red neuronal que consistía en la unión de diferentes perceptrones donde un científico calculaba manualmente el peso de cada uno de los perceptrones, siendo una tarea muy difícil y de poca eficacia en sistemas con un gran número de perceptrones.

### 2.3.2 Etapa 2. Conexionismo (años 1980 – 1995)

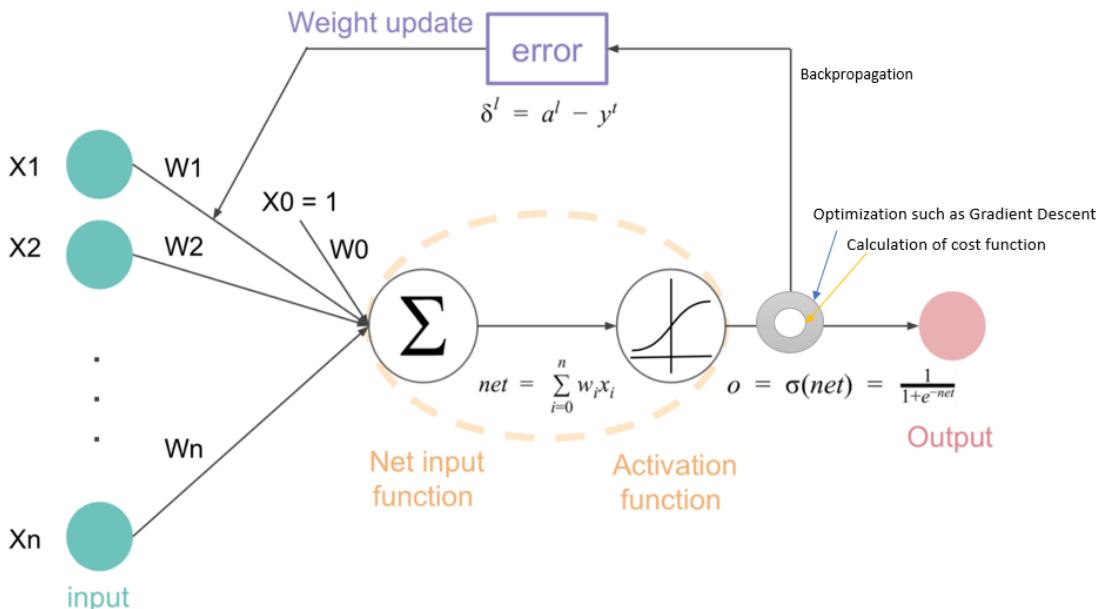
En esta época Rummelhart introdujo en 1986 el concepto de *backpropagation* [Rummelhart et al, 1986] que actualmente se utiliza en el entrenamiento de la mayoría de las redes neuronales para recalcular el peso de cada una de las neuronas que componen una determinada estructura de red.

Para solucionar el problema de tener que calcular los pesos manualmente, en los años 80 se creó la primera neurona con una función de activación que permitía devolver valores reales en lugar de 0 y 1 como se trabaja inicialmente.

Esta función de activación fue la sigmoidal y supuso un gran cambio a la hora de trabajar con estas redes, ya que se podían lograr pequeñas alteraciones en los valores de los pesos para ir produciendo pequeñas alteraciones en la salida, permitiendo ajustar poco a poco los pesos de las conexiones obteniendo los valores correctos.

Con la creación del algoritmo *backpropagation* se hizo posible el entrenamiento de las redes neuronales con múltiples capas, de forma supervisada y calculando el error obtenido en la salida.

Su funcionamiento era propagar el error hacia las capas anteriores realizando pequeños ajustes en cada iteración para lograr que la red aprenda de forma automática.



**Figura 9:** Funcionamiento del algoritmo Backpropagation sobre un perceptrón

En 1989 se crearon las redes neuronales convolucionales inspirándose en el córtex visual de los animales. La primera implementación fue generada por Yann LeCun y estaba enfocada en el reconocimiento de las letras manuscritas [[Yann LeCun et al., 1989](#)].

La estructura constaba de varias capas que implementaban la extracción de características y luego las clasificaban. La imagen de entrada alimenta una capa convolucional la cual se encarga de extraer las características de la entrada, como la detección de líneas o vértices.

El siguiente paso se denomina *pooling*, el cual reduce las dimensiones de las características extraídas intentando mantener la información relevante. Estas dos etapas se repiten varias veces hasta obtener el tamaño de nuestro interés.

Una vez que han finalizado las etapas de convolución y *pooling* se alimenta una red multicapa donde la salida final de la red es un grupo de nodos que clasifican el resultado, por ejemplo, un único nodo que si su valor es 0 corresponde a una imagen de la clase 1 y si es un 1 corresponde a una imagen de la clase 2.

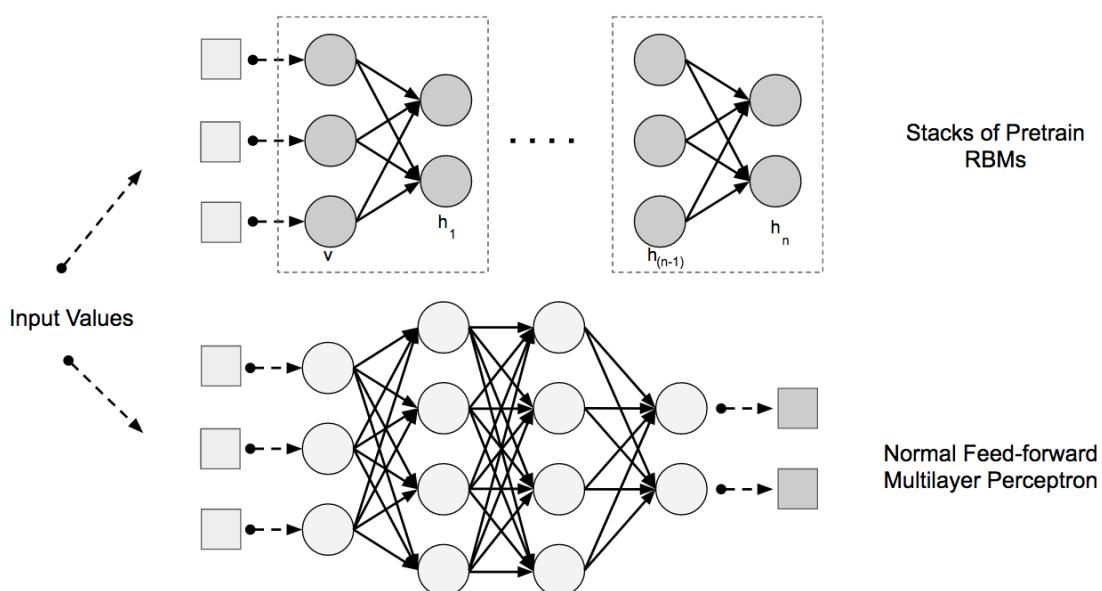
### 2.3.2 Etapa 3. Deep Learning (años 2006 – actualidad)

Lo que actualmente estamos viviendo arrancó principalmente a raíz de estos tres documentos: [[Hinton et al, 2006](#); [Bengio et al, 2007](#); [Ranzato et al, 2007](#)].

Históricamente los principales modelos pretendían emular el aprendizaje biológico como hemos visto en el caso del perceptrón, llegando a asociar este tipo de estudio al Deep Learning. Sin embargo, a pesar de que los modelos constituidos con este tipo de estructura están inspirados en el cerebro biológico, no están diseñados para hacer una representación realista de este.

El primer hito surgió con las redes *Deep Belief Networks (DBN)* que intentaban mejorar los modelos de redes profundos con varias capas de neuronas difíciles de entrenar aun usando las técnicas de propagación hacia atrás. Estas redes consistían en utilizar un [Autoencoder](#) con [Restricted Boltzmann Machines](#) para preentrenar a las neuronas de la red y obtener un resultado mejor.

Este tipo de red demostró que el iniciar aleatoriamente los pesos de la red no da los mejores resultados ya que puede llegar a caer en mínimos locales. Utilizando las técnicas propuestas por las DBN, entrenando inicialmente de forma no supervisada con el uso del *Autoencoder*, y aplicando posteriormente las técnicas globales de *backpropagation*, consiguieron mejores resultados que con la forma tradicional.



**Figura 10:** Estructura de una red Deep Belief Networks

Actualmente se siguen usando este tipo de estructuras para resolver problemas complejos y abordar otro tipo de problemas nunca vistos, sobre todo basados en la automatización, como veremos en el siguiente punto.

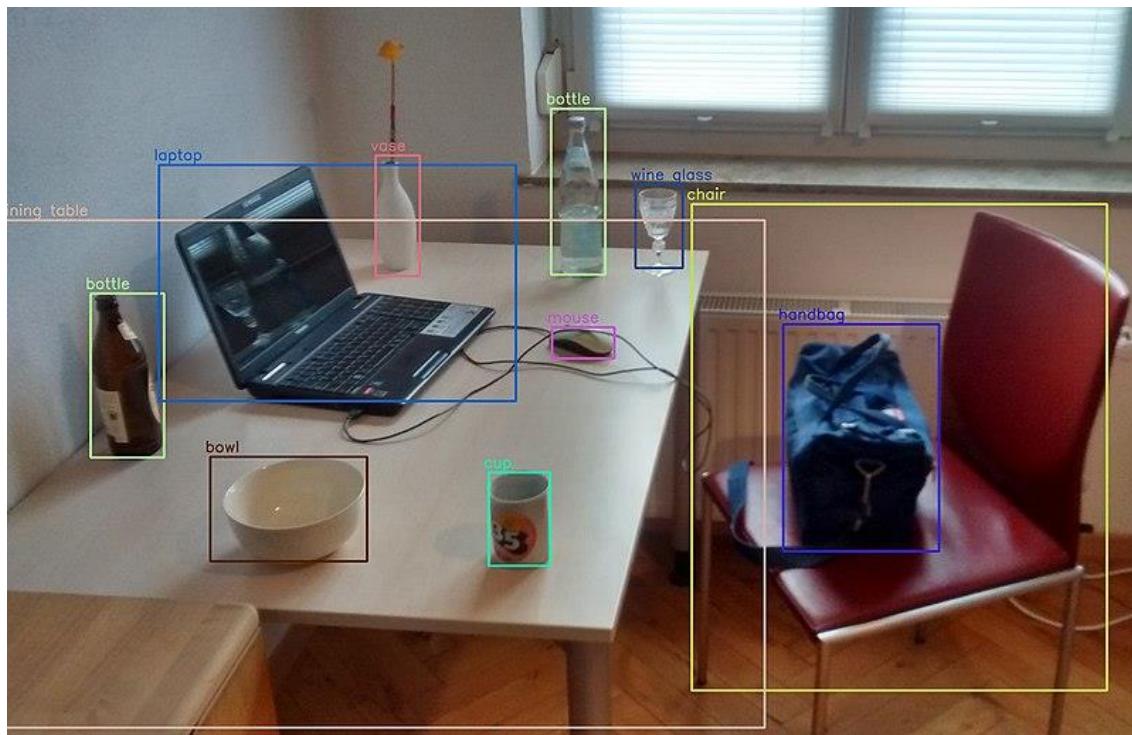
## 2.4 Estado Actual

---

En la actualidad el Deep Learning es una de las herramientas que más usos tiene con respecto a los diferentes tipos de problemas que nos podemos encontrar.

Este fenómeno se debe a que este tipo de aprendizaje trata de imitar acciones humanas que nosotros desempeñaríamos en pocos segundos en computadores de grandes capacidades en tareas como el reconocimiento de imágenes.

El reconocimiento de imágenes ha sido posible gracias a la creación de las redes convolucionales que imitan el funcionamiento del córtex de los animales, semejándose al funcionamiento de nuestro sistema biológico para reconocer las diferentes imágenes y consiguiendo identificar diferentes objetos dentro de la misma.



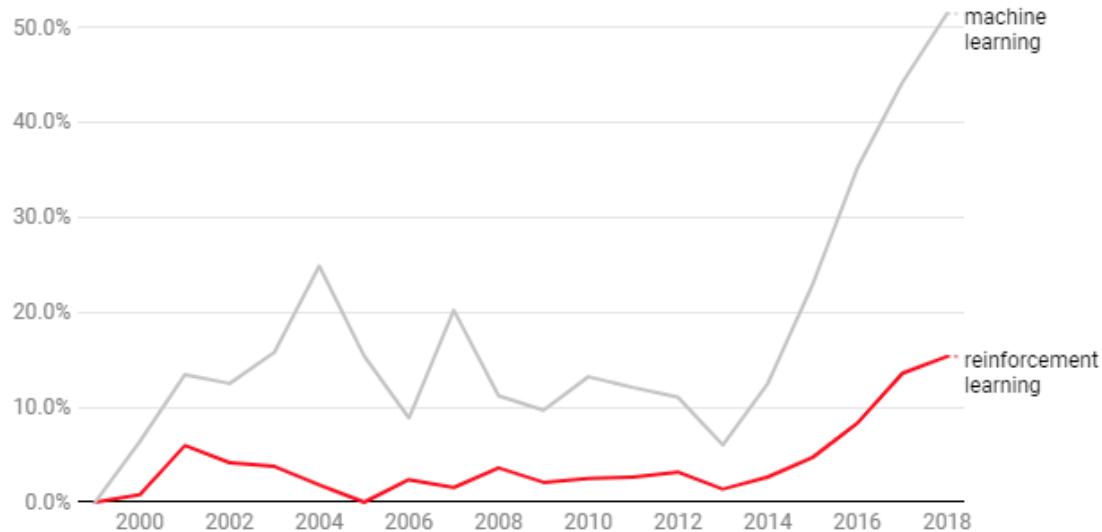
**Figura 11:** Ejemplo de reconocimiento de objetos en una imagen

Uno de los principales factores que ha llevado a este tipo de tecnología al auge ha sido el avance de la tecnología, permitiendo realizar grandes procesamientos de datos en poco tiempo gracias al paralelismo masivo y a la gran capacidad de computo de los nuevos procesadores y tarjetas gráficas. Estas últimas cuentan con centenares de procesadores capaces de tratar miles de imágenes con multitud de píxeles en un tiempo razonable para una aplicación informática.

Al hacerse posible la implementación de estas tecnologías tan potentes en prácticamente cualquier ordenador, los usuarios han tenido la capacidad de investigar diferentes métodos de trabajo dentro de este mundo, que, sumado a la liberación del *framework* TensorFlow de Google, ha hecho que el número de tutoriales y publicaciones haya subido de forma importante, haciendo más accesible el entendimiento de este tipo

de contenido a cualquier persona que esté interesada en aprender un poco más sobre esta tecnología.

En la figura 12 podemos ver el incremento de publicaciones que hablan sobre el aprendizaje automático en los últimos años, notando la popularidad de este.



**Figura 12:** Porcentaje de artículos sobre Machine Learning y Aprendizaje por Refuerzo

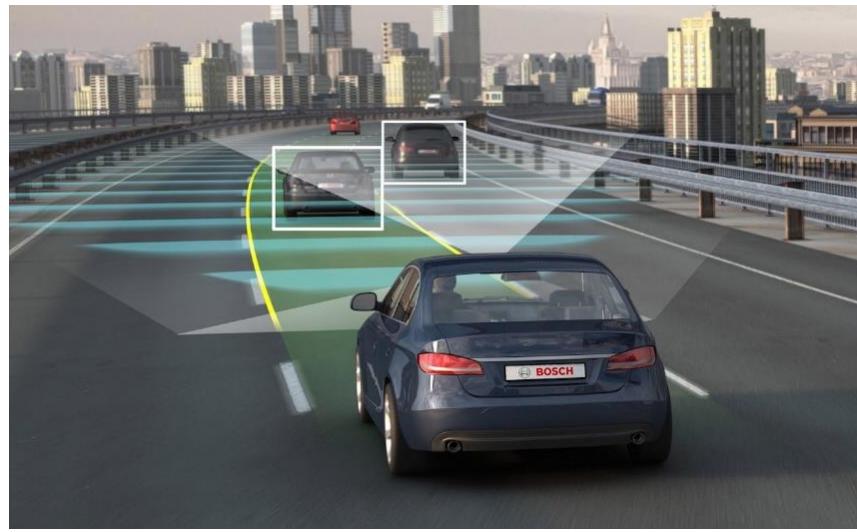
Uno de los sectores donde el *Machine Learning* ha tenido más repercusión ha sido en el sector del marketing, donde se ha convertido en el principal recurso de las empresas para el estudio de datos de los clientes y la elaboración de recomendaciones a partir de ellos.

Grandes compañías, como Netflix, recomienda programas basados en las preferencias de los usuarios según los datos demográficos y el historial de cada espectador, haciendo que casi un 75% de las series que veamos los usuarios de Netflix sean recomendaciones de la propia aplicación.

El Aprendizaje Automático ha tenido a su vez una gran importancia en problemas que inicialmente parecían imposibles de resolver, como en el sector de la automoción, donde inicialmente construyeron automóviles capaces de aparcar solos y, en la actualidad, existen capaces de conducir de manera autónoma con los algoritmos de *Machine Learning*.

Actualmente existe una carrera comercial por parte de las principales compañías automovilistas por conseguir el primer coche capaz de trabajar sin conductor y comenzar así una nueva revolución tecnológica.

El uso de esta tecnología en algo tan cotidiano como conducir plantea otro tipo de problemas como los de responsabilidad en caso de accidente o el concepto de aprendizaje. Como hemos visto, este tipo de aprendizaje se basa en ejemplos, y, en la mayoría de los casos, no podemos estudiar todas las posibles situaciones con experimentación, quedando algunas situaciones a la suerte, que en este caso puede suponer la muerte de una persona.

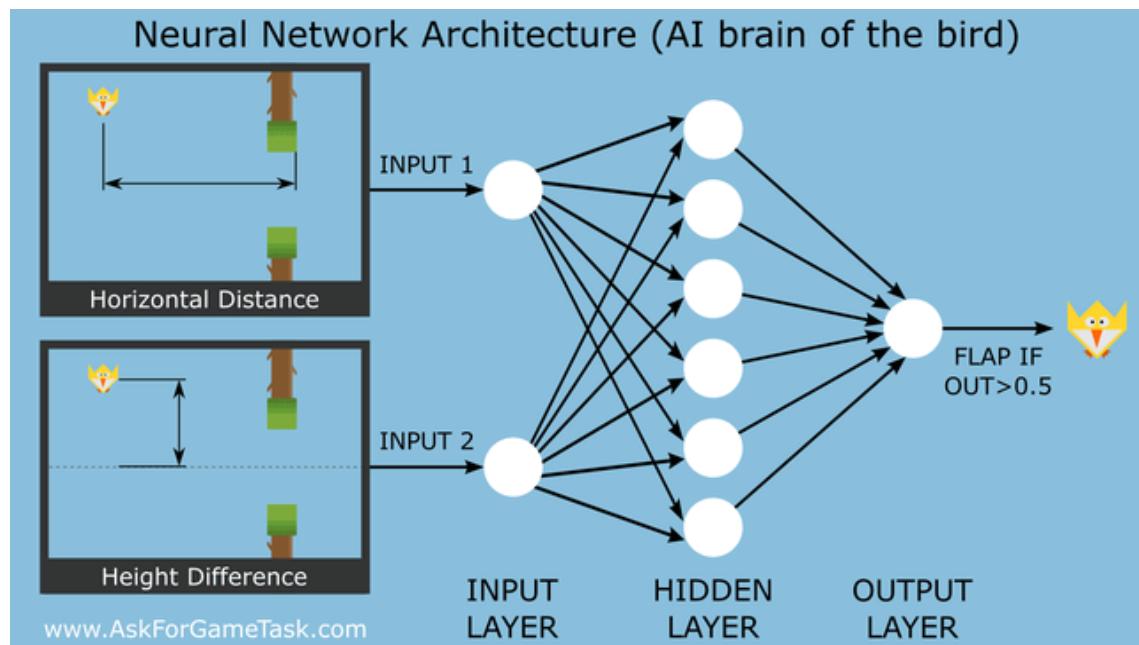


**Figura 13:** Coche percibiendo del entorno con el uso de sensores

Otro sector donde el Machine Learning está tomando una gran importancia es en el mundo de la automatización con robots que son capaces de aprender de sus propias acciones o de lo que perciben del entorno.

Al incorporar esta tecnología conseguimos simular comportamientos parecidos al de los humanos y haciendo, tanto a los robots físicos, como a la inteligencia artificial usada en los videojuegos, más realistas y, por tanto, más visibles para el público.

La automatización en este sentido nos puede llevar a pensar que el futuro estará compuesto por robots capaces de comportarse como humanos y por lo tanto capaces de sustituirnos en ciertos sectores, como ya ha sucedido en la industria con la automatización de los procesos industriales.



**Figura 14:** Automatización de un personaje de un videojuego mediante redes neuronales

En este trabajo nos centraremos en el uso de esta tecnología en el campo de la visión por computador, donde la usamos para el procesamiento de las imágenes. En este sector el algoritmo más importante que nos encontramos es el CNN, ya que es el más eficaz para el tratamiento de las imágenes, siendo capaz de obtener información de estas con gran precisión.

En este campo actualmente nos encontramos multitud de procesos que ya están automatizados gracias al reconocimiento de imágenes, como en el control de calidad de los productos fabricados de forma masiva o el control por cámaras de seguridad en algunos aeropuertos, donde un programa de *Machine Learning* es capaz de reconocer criminales con tan solo visualizarlos en una imagen a tiempo real.

Este tipo de algoritmos ha demostrado tener un gran potencial en multitud de los principales problemas de actualidad, consiguiendo multitud de hitos en el mundo de la inteligencia artificial que nos acercan poco a poco a la sociedad del futuro, donde la mayoría de los procesos lleguen a estar automatizados y la vida sea más fácil.



**Figura 15:** Reconocimiento de personas mediante técnicas de Machine Learning

---

# Capítulo 3

## Fundamentos teóricos

---

Anteriormente hemos hecho un primer acercamiento a los diferentes recursos que vamos a usar durante la ejecución de la memoria en un formato muy general y ejemplificado, dejándonos claro el gran potencial que tiene este tipo de algoritmos y el futuro de estos con el uso de las placas introducidas en el apartado anterior.

En esta sección de la memoria vamos a dar un paso más sobre los conocimientos previos a la experimentación, necesarios para comprender en profundidad el funcionamiento de las redes neuronales convolucionales, que será el principal recurso necesario para desarrollar el modelo que posteriormente portaremos en la placa Maix Bit.

Una vez que conozcamos todos los detalles sobre este tipo de algoritmos, pasaremos a conocer las diferentes tecnologías que usamos en el trabajo, remarcando la importancia de cada una de ellas y justificando el uso y las ventajas que pueden aportar a nuestro trabajo.

Por último, hablaremos de los diferentes pasos para la construcción del modelo y su posterior traslado al dispositivo portable, detallando de forma teórica los diferentes pasos necesarios para su correcto funcionamiento y ejecución.

---

### 3.1 Redes Neuronales Convolucionales

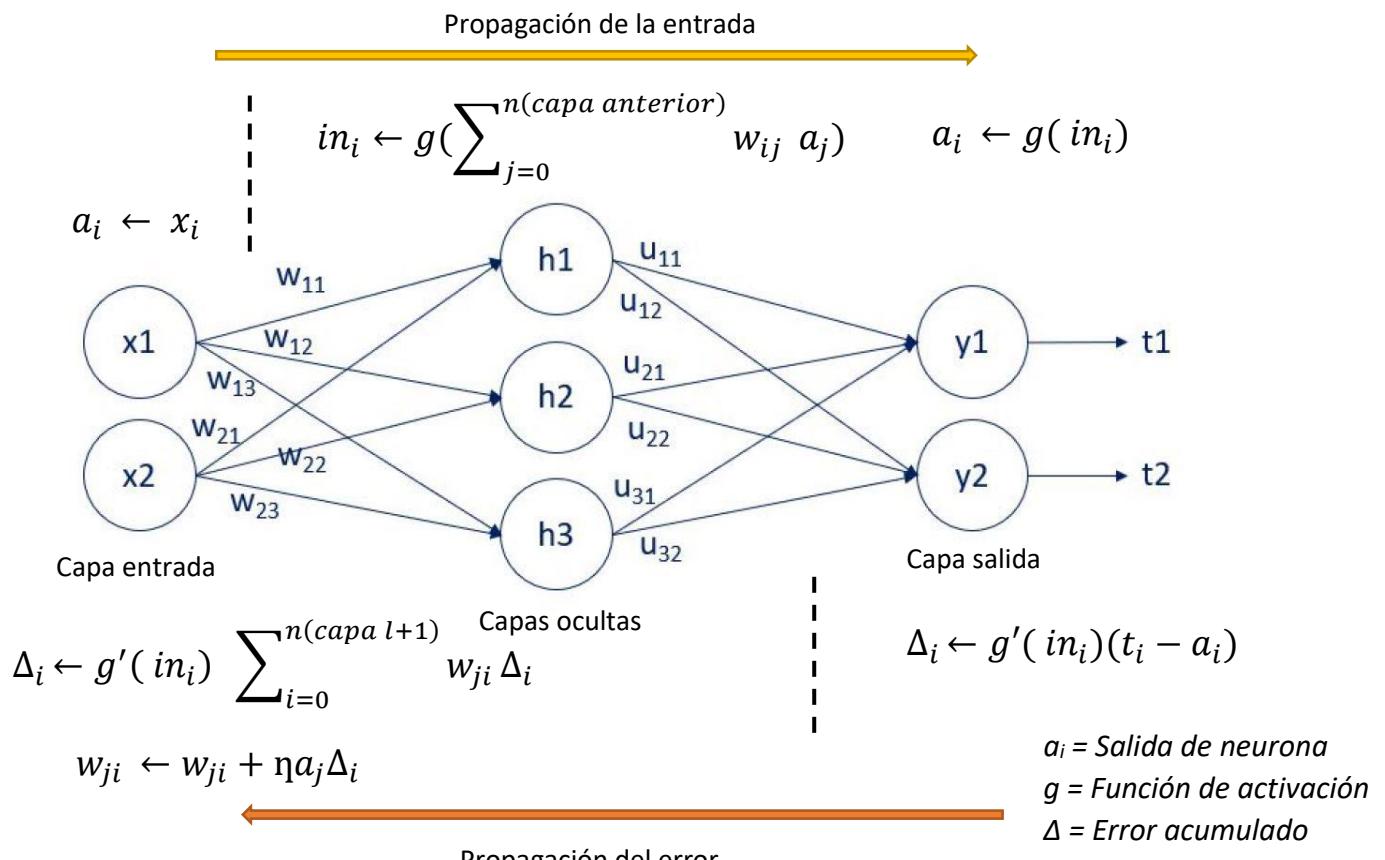
---

Como hemos visto en el capítulo anterior, existe un tipo de red neuronal artificial que se basa en la recreación del córtex visual en los mamíferos. Mediante el uso de neuronas artificiales somos capaces de obtener automáticamente información relevante de las matrices de las que están formadas las imágenes.

En las redes neuronales clásicas, normalmente trabajamos con diferentes neuronas conectadas entre sí en una jerarquía de capas donde podemos diferenciar: la capa de entrada, donde se introducen los valores a analizar por la red neuronas; las capas ocultas, que se caracterizan por estar compuestas de una red completa de neuronas que automáticamente irán ajustándose para llegar a la solución final; y por último la capa de salida, formada por tantas neuronas como posibles soluciones sean posibles.

La forma de entrenar este tipo de redes consiste en el cálculo continuo de los pesos asociados a cada una de las transiciones entre neuronas de forma que, para los valores de entrada en un momento determinado, el error generado a la salida (que es conocida al tratarse de aprendizaje supervisado) sea mínimo.

Una de las formas más conocidas para el entrenamiento de la red es la conocida como *Backpropagation*, la cual una vez que ha calculado la salida de cada una de las neuronas y ha obtenido el error cometido con respecto al valor real en la salida, hace un recorrido “hacia atrás” traspasando el error a cada transición de neuronas y recalculando el valor de los pesos respecto a este.



**Figura 16:** Resumen y estructura algoritmo Backpropagation

Las CNNs con muy similares a este tipo de redes que acabamos de ver, tienen su propia estructura formada por varias neuronas con sus pesos y salidas. Esta estructura de neuronas se divide en un tipo de capas diferentes a las clásicas que especificaremos en el siguiente apartado [3.2](#) de este capítulo.

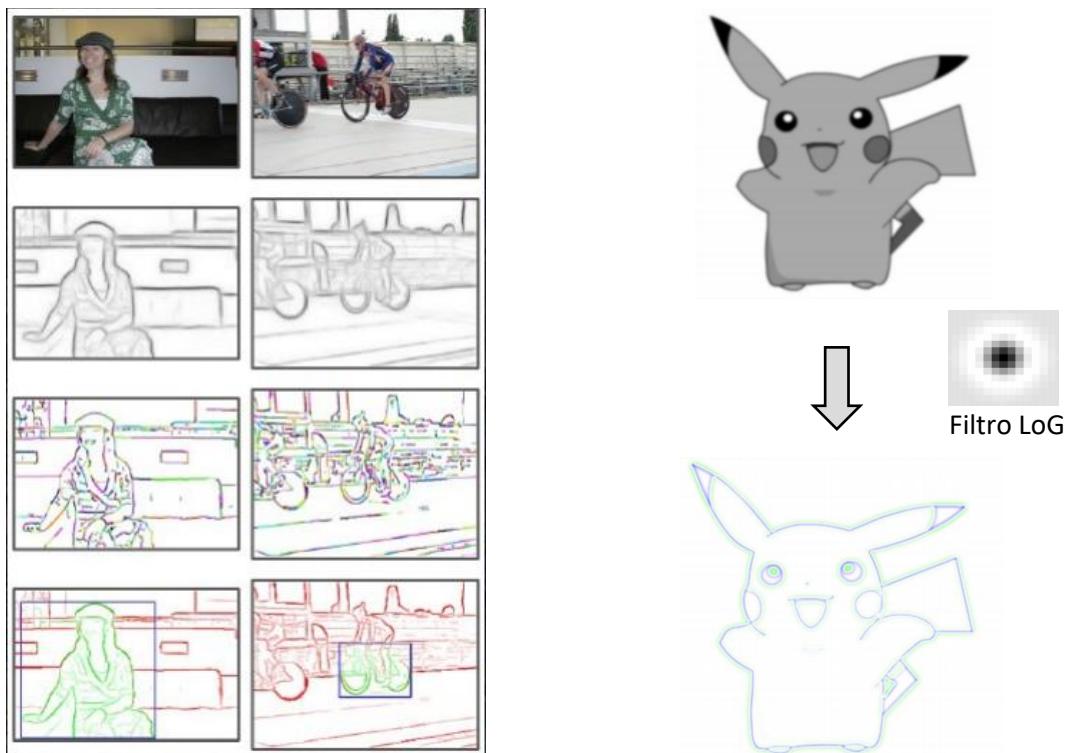
La introducción de nuevas capas a la estructura de red nos convierte nuestra estructura en una herramienta mucho más versátil para cierto tipo de operaciones como los relacionados con la visión artificial, las cuales vamos a tratar en este proyecto.

Una de las diferencias que más pueden destacar entre las redes clásicas y las CNNs puede atribuirse a los datos de entrada de cada una de ellas, ya que, para las clásicas, un experto era el responsable de encontrar las características relevantes de un conjunto de datos, mientras que, en las convolucionales, es el propio algoritmo el encargado de encontrar estas características de los datos sin tratar.

Uno de los principales ejemplos se da en la clasificación de imágenes, ya que, se tratan de matrices numéricas las cuales contienen la información de cada uno de los píxeles que la componen de forma individual, haciendo casi imposible su estudio por medio de las redes neuronales clásicas por su dificultad al tratar de encontrar ciertas características.

Sin embargo, las redes convolucionales tienen una estructura preparada para el entrenamiento con datos de grandes dimensiones y multitud de posibles características. Estas redes son capaces de obtener características de interés de las imágenes de entrada con el uso de diferentes máscaras (Kernel), tales como detección de rectas o contornos que se asociarán posteriormente a la salida correspondiente.

Esta extracción de características se produce en las capas de convolución, que, combinada con otro tipo de operaciones, son capaces de ir aprendiendo a diferenciar desde colores o pequeños rasgos al inicio de la red hasta detalles mucho más complejos como pueden ser los ojos y la boca en un retrato.

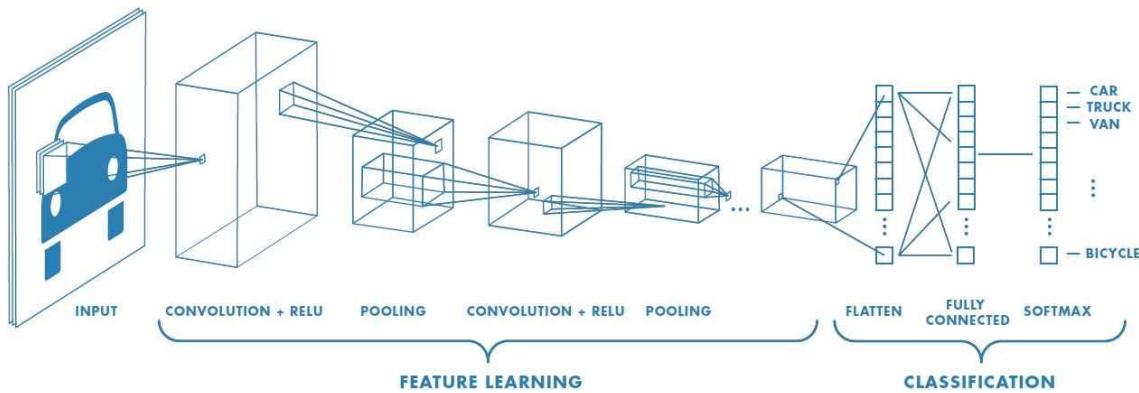


**Figura 17:** Imágenes tratadas por diferentes capas de convolución

Con estos dos ejemplos podemos suponer la importancia de encontrar un filtro adecuado para cada una de las imágenes de entrada. El entrenamiento de este tipo de redes incluye el cálculo automático de estos filtros de forma que se obtenga el mayor número de características posibles.

Debido a la gran magnitud de datos que se obtienen de las imágenes de entrada y de las diferentes operaciones de convolución (aplicación de máscaras por toda la imagen), se aplica una capa de *pooling* dedicada a reducir el tamaño de los datos y por tanto el número de neuronas en cada capa, haciendo que su entrenamiento sea mucho más rápido que si mantuviéramos todas las neuronas iniciales y asegurándose de mantener las características obtenidas en la capa anterior.

Una vez aplicadas todas las capas de convolución y *pooling* se introducen los datos obtenidos como entrada de una red neuronal clásica, la cual es la que se va a encargar de distinguir entre los diferentes tipos de clases en el caso de los clasificadores de imágenes.

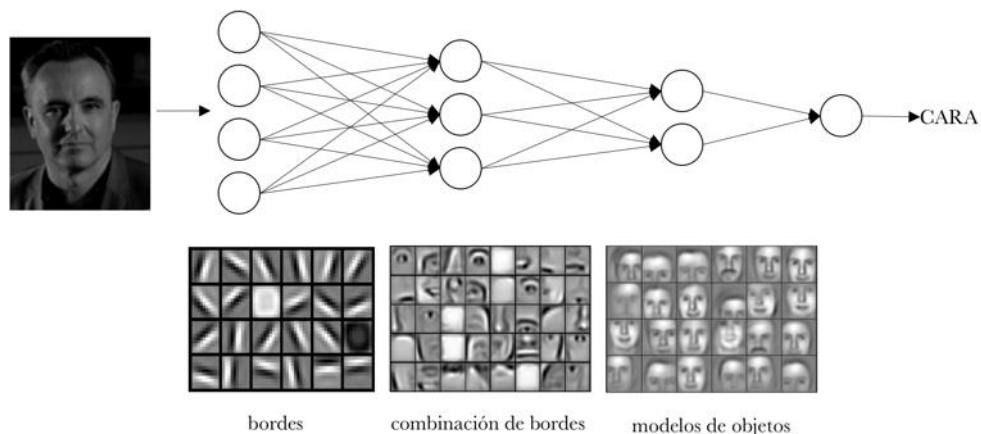


**Figura 18:** Estructura de una red neuronal convolucional

El entrenamiento de la estructura de red completa es similar al de todas las redes neuronales existentes y es que se basa en la minimización del error con respecto a la salida obtenida.

Este tipo de cálculo puede desarrollarse con multitud de algoritmos diferentes y dependerá de la experimentación individual por parte del ingeniero en encontrar el óptimo para su problema.

La estructura de red forma parte de esa experimentación y es que, dependiendo del número de capas y del número de neuronas por cada una de ellas, podemos obtener resultados muy distintos unos de otros. Por ello, el trabajo de experimentación tiene una gran importancia en este tipo de problemas en los que es el propio algoritmo el encargado de clasificar automáticamente las imágenes en función de sus características.



**Figura 19:** Red CNN con las características obtenidas en cada capa de forma esquemática

La elección de la estructura y de las imágenes de entrada, así como la elección del criterio de *pooling* y de cálculo de error, serán justificados a lo largo del proyecto junto al estudio de los resultados obtenidos.

## 3.2 Tipos de capas en las redes CNN

---

Las redes neuronales convolucionales están formadas por multitud de capas diseñadas para efectuar las diferentes tareas necesarias para realizar tanto el entrenamiento de la red como la optimización de esta.

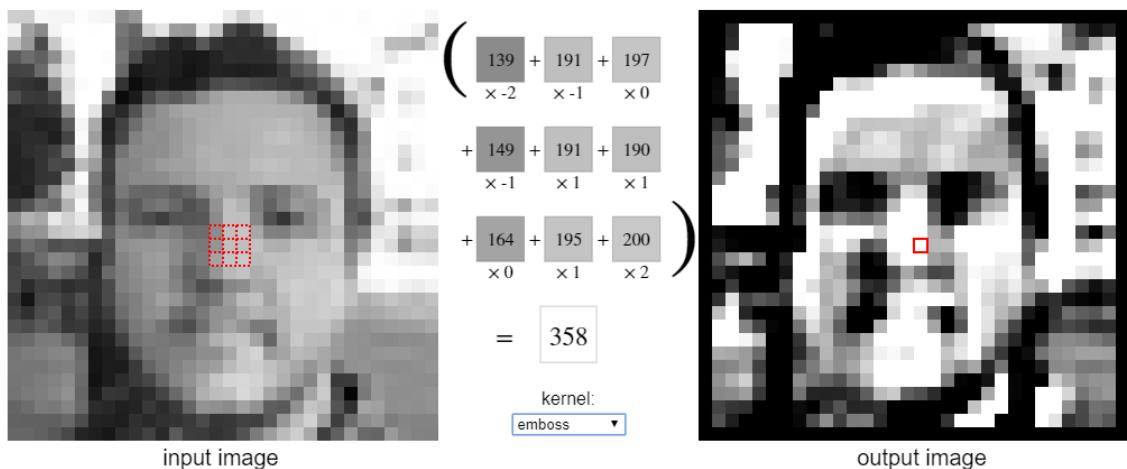
Existen multitud de operaciones que se pueden evaluar en este tipo de redes, desde el cálculo del error hasta la obtención de algún parámetro concreto necesario para el problema en cuestión. Cada una de las capas que pueden ser usadas tiene unas características y una función propia que la diferencia de las demás. A continuación, explicaremos y ejemplificaremos algunas de las capas más usadas.

### 3.2.1 Capa de Convolución

---

La capa de convolución es la encargada de extraer las características de las matrices que recibe como entrada, que pueden ser la propia imagen original o la salida de alguna de las otras capas. La salida resultante de esta capa es llamada mapa de características.

La forma de conseguir estas características a partir de los datos es con el uso de filtro o *kernels* especializados en encontrar diferentes tipos de características, como pueden ser los dedicados a la obtención de bordes o al difuminado de fondos. En la página de [Setosa](#) podemos visualizar algunos de los filtros más usados con ejemplos y resultados tras su aplicación, como podemos ver en la figura 20.



**Figura 20:** Ejemplo filtro emboss sobre una imagen

La operación de convolución es una operación matemática que consta de la transformación de dos matrices ( $f$  y  $x$ ) en una tercera que representa en cierto sentido la magnitud en la que se superponen  $f$  y una versión modificada de  $x$ . En nuestro caso,  $f$  se corresponde con la matriz de entrada y  $g$  con el filtro a aplicar a la imagen.

Según el profesor [Jianxin Wu](#), como publica en su publicación sobre la introducción a las redes neuronales [[Jianxin Wu, 2017](#)], la convolución matemáticamente se expresa con la siguiente ecuación:

$$y_{i^{l+1},j^{l+1},d} = \sum_{i=0}^H \sum_{j=0}^W \sum_{d=0}^{D^l} f_{i,j,d^l,d} \times x_{i^{l+1}+i,j^{l+1}+j,d^l}^l .$$

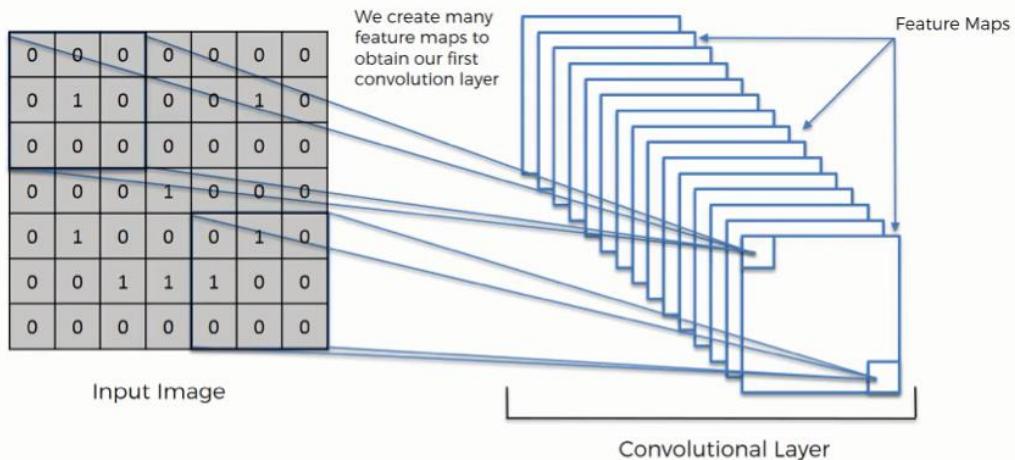
**Figura 21:** Fórmula matemática de la convolución

En la figura [21](#) podemos diferenciar las matrices a transformar como ‘ $f$ ’ y ‘ $x$ ’ y la matriz resultante como ‘ $y$ ’ que se forma con el sumario de la convolución efectuada en todo el espacio de la matriz ‘ $f$ ’, donde  $H$  es la altura,  $W$  es la anchura y  $D$  son las dimensiones en las que está dividida la matriz entrante.

Esta operación genera una matriz de características con el resultado de haber aplicado los filtros correspondientes, teniendo cada una de ellas unos datos representativos diferentes que serán elegidos mejores características con el entrenamiento de nuestra red.

La elección de estos filtros es automática durante el entrenamiento, ya que la etapa de convolución no se aplica únicamente a un filtro, sino que se aplica a un conjunto de ellos, y es el propio algoritmo el que va seleccionando cuales de ellos devuelven mejores características para su posterior estudio. De esta forma, el algoritmo puede llegar a elegir filtros que para el ojo humano no tienen sentido o no parecen relevantes, pero, sin embargo, para la máquina son los que mejor diferencian a las diferentes clases.

Aunque todo parezca automático, es trabajo del propio programador en elegir el tamaño de las imágenes de entrada, así como el tamaño del filtro a utilizar y la forma que tiene de aplicarse los filtros, ya que hay veces que no interesa aplicar a toda la matriz entrante y solo nos interesan ciertas zonas de ellas.



**Figura 22:** Salida de la capa de convolución

## Operación ReLU (*Rectified Linear Unit*)

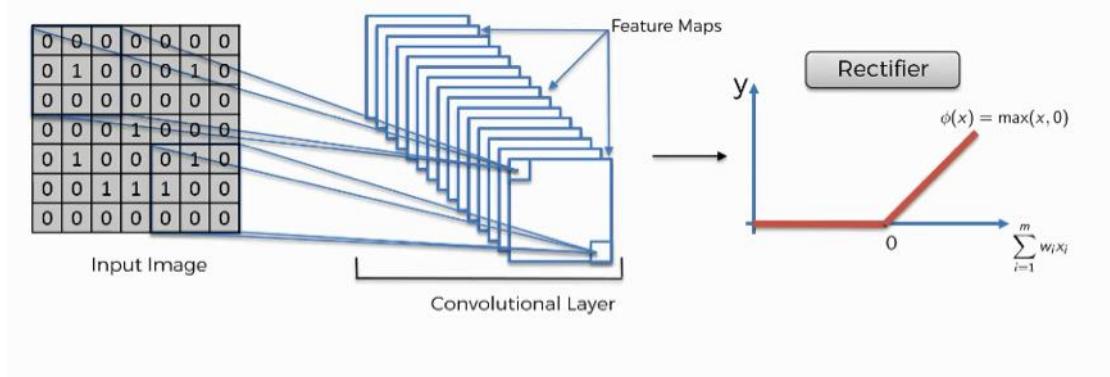
Este tipo de operación normalmente se incluye dentro de la capa de convolución y sirve como un paso complementario al paso de convolución anterior. Esta operación sirve como rectificador para incrementar la no linealidad de las imágenes.

Todas las imágenes son no lineales por naturaleza, tenemos claro que en ellas se incluyen elementos tales como la transición entre píxeles, bordes, colores... que hacen que nuestra imagen presente irregularidades en sus gráficas de píxeles.

Pero, si las imágenes que introducimos son no lineales, ¿por qué deberíamos intensificarlo más? Esto se realiza debido a que en la capa de convolución podríamos perder esos elementos que para nuestro problema son tan valiosos, como los bordes, debido a que se produce un cambio de tamaño y una aplicación de un filtro que podría llegar a suavizar la imagen.

Esta rectificación se designa como función de activación y a veces puede ser vista como una capa más en el algoritmo CNN, aunque siempre se produce tras aplicar la convolución, por lo tanto, podemos tratarla como una única capa.

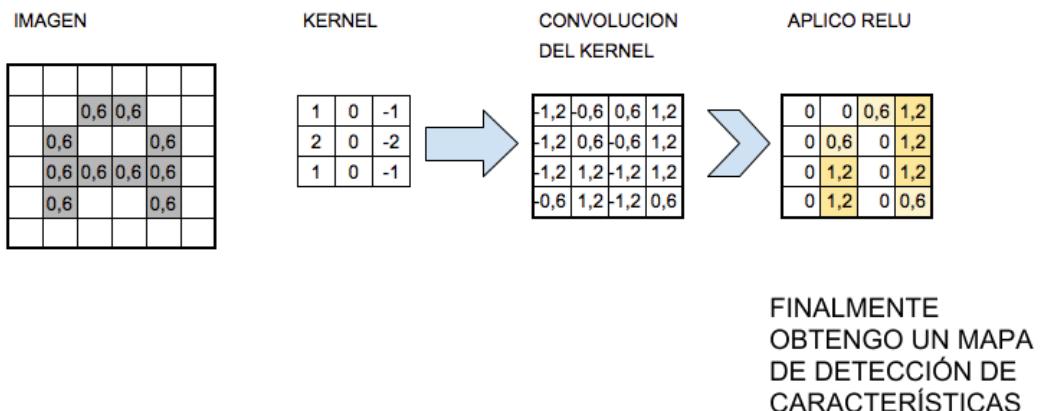
La operación ReLU en concreto utiliza como función de activación el máximo entre 0 y el valor de una tupla en concreto, eliminando los elementos negativos de la matriz.



**Figura 23:** Capa de convolución + ReLU

Existen dos documentos donde se ejemplifican de forma mucho más profunda tanto los conceptos matemáticos como la justificación de uso de estos rectificadores. El primero se le corresponde al profesor de la universidad de California Jay Kuo [[Jay Kuo 2016](#)], y el segundo a un grupo de investigación de la empresa Microsoft, donde aparece el estudio de diferentes operaciones de rectificación [[Kaiming He et al, 2015](#)].

Como resumen de operación de convolución, en la figura 24, podemos ver un ejemplo de los diferentes pasos dados en la etapa con una imagen con unos valores de prueba que nos da una visión menos teórica de esta etapa.



**Figura 24:** Resumen ejemplificado de la etapa de convolución

### 3.2.2 Capa de Muestreo o Agrupación (*Pooling*)

Esta etapa se aplica a continuación de la capa de convolución, tomando sus mapas de características como entrada. Los principales objetivos de esta etapa son tanto mantener las principales características de las diferentes imágenes como reducir el tamaño de las matrices que vamos a usar.

Si al aplicar la convolución hemos usado 32 filtros en imágenes de 32x32 píxeles, obtendremos 32 mapas de características de un tamaño similar o igual al de la imagen de entrada, necesitando una gran cantidad de neuronas para su estudio y ralentizando el proceso de entrenamiento y clasificación en gran medida.

La búsqueda por mantener las principales características tiene un sentido obvio ya que el trabajo final de nuestro algoritmo es encontrar las principales características entre las imágenes de las diferentes clases en cualquier situación. Por ejemplo, si estamos intentando aprender que imágenes son de un velero, deberemos suponer que el velero puede estar atracado en el puerto o navegando; que la imagen puede tener rotación o cambio de escala; u otras muchas características que hacen a la imagen diferente, aunque pertenezcan a la misma clase.

Con el uso del *Pooling* conseguimos que el algoritmo aprenda a diferenciar las imágenes a pesar de las diferentes situaciones en las que se encuentre, aprendiendo a detectar las características comunes entre las imágenes de la misma clase independientemente de la situación del objeto en la matriz.

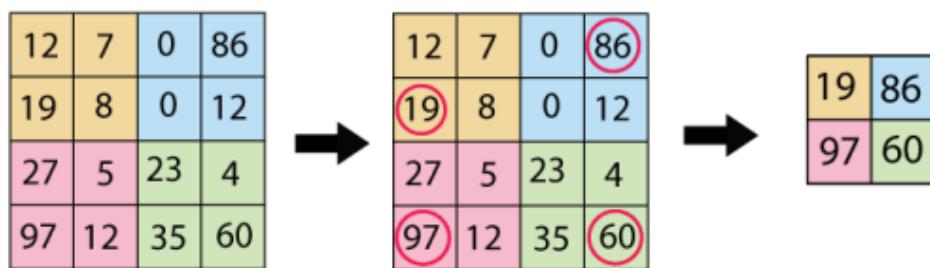


**Figura 25:** Diferentes posibles imágenes de entrada

Podemos encontrarnos diferentes tipos de *Pooling*, aunque nosotros nos vamos a centralizar en el método de *Max Pooling*, que es uno de los más usados en los problemas de clasificación de imagen.

Este proceso consiste en recorrer toda la matriz de características, pero en lugar de ir eligiendo un único píxel, elegiremos tantos como el tamaño que queramos aplicar a esta capa. Por ejemplo, si queremos usar un tamaño de  $2 \times 2$ , iremos recorriendo la matriz entrante por agrupaciones de 4 píxeles, donde nos elegiremos únicamente con el píxel de mayor valor, descartando el resto.

## Pooling—Max pooling

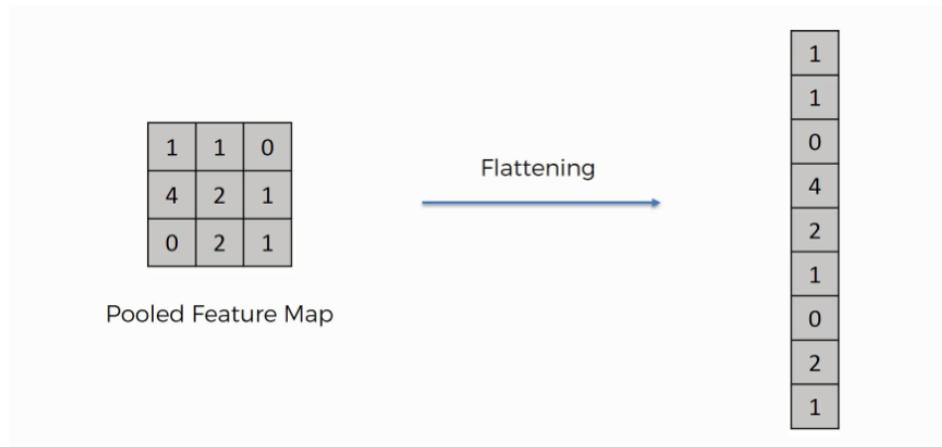


**Figura 26:** Ejemplo Max Pooling  $2 \times 2$

Los diferentes parámetros que debemos decidir a la hora de la construcción de la red son parecidos a los de la capa anterior, atendiendo a las diferentes posibilidades de tamaño o tipo de agrupación a usar, aunque mayoritariamente es usado la agrupación de máximos en tamaños no superiores a  $3 \times 3$  píxeles. En el documento presentado en la ICANN de 2010 se muestra el estudio en más profundidad de los diferentes modelos de *pooling* junto a su evaluación con diferentes ejemplos [[Dominik Scherer et al, 2010](#)].

### 3.2.3 Capa de Aplastamiento o *Flattening*

Una vez que se han efectuado varias capas de convolución y *pooling* obtenemos un gran número de matrices de características que deben ser evaluadas para su posterior clasificación. El objetivo de esta etapa es transformar la salida obtenida de la última capa de *pooling* en un vector con los valores de cada una de las matrices, que se convertirá en la entrada de la red neuronal de la siguiente capa.



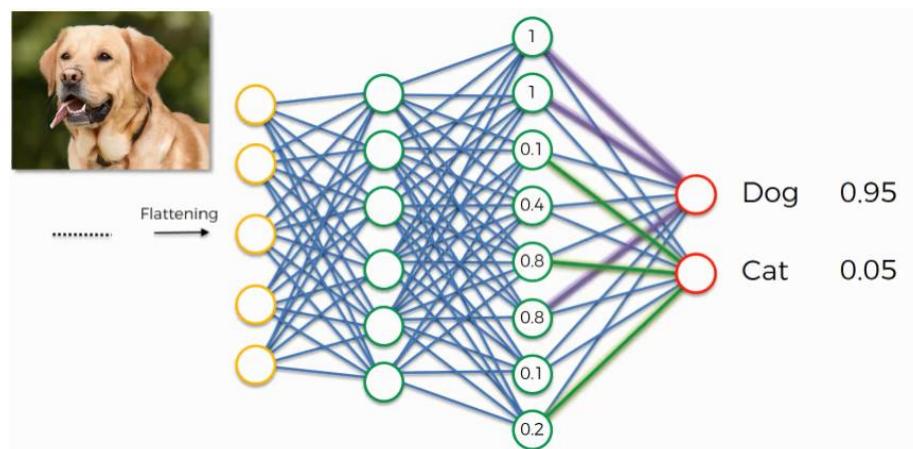
**Figura 27:** Ejemplo aplastamiento de una matriz

### 3.2.4 Capa de *Fully-Connected*

En esta capa es donde las redes neuronales convolucionales y las redes neuronales artificiales se conectan de forma que la entrada a esta es la salida de la capa de aplastamiento y la salida de la red es el resultado obtenido del estudio de la imagen inicial.

El objetivo de la red neuronal tradicional es asociar las características entrantes a sus clases correspondientes con el ajuste de los pesos de los enlaces de sus neuronas con el cálculo del error.

A diferencia de las redes tradicionales donde usualmente se usaban como cálculo del error la función coste o la media del error cuadrático, en las CNN es más frecuente referirnos al error como la función de perdida, la cual refleja cuanto de bueno son las características con la clase a la que deben referirse.



**Figura 28:** Ejemplo de clasificación de una imagen de un perro

La salida de nuestras redes no devuelve un resultado 100% exacto en el criterio de clasificación ya que en las imágenes normalmente siempre hay características que se repiten y por lo tanto son comunes para las diferentes clases. Los resultados usualmente son porcentajes con la probabilidad de que sea una clase u otra en relación con las características de la imagen entrante.

Existen multitud de posibilidades de red dentro de las tradicionales, teniendo que ajustarnos a los parámetros y a la magnitud de nuestro problema, ya que no es lo mismo clasificar imágenes de un tamaño que de otro. Normalmente las imágenes tratadas en este tipo de algoritmo son imágenes de baja resolución, ya que son suficientes para su clasificación y suponen un mejor rendimiento y un entrenamiento más rápido.

En resumen, las diferentes capas que contiene el algoritmo están especialmente diseñadas para el estudio de las imágenes, con la búsqueda de características que posteriormente será aplicado a la red neuronal clásica, la cual se encargará de obtener los resultados.

La gran cantidad de información que recibe nos obliga a trabajar con procesadores o incluso con las GPUs para ser capaces de realizar todos los cálculos necesarios en el algoritmo en un tiempo razonable de entrenamiento, ya que el número de imágenes necesarias para conseguir una red entrenada que de buenos resultados normalmente es una cifra muy elevada.

### 3.3 Introducción a las tecnologías usadas

Una vez visto el algoritmo que vamos a usar para la primera parte del trabajo, vamos a hacer un repaso a las diferentes tecnologías implicadas en la segunda parte con intención de justificar el uso de este tipo de placas junto las diferentes ventajas a la hora de llevarlo al ámbito profesional.

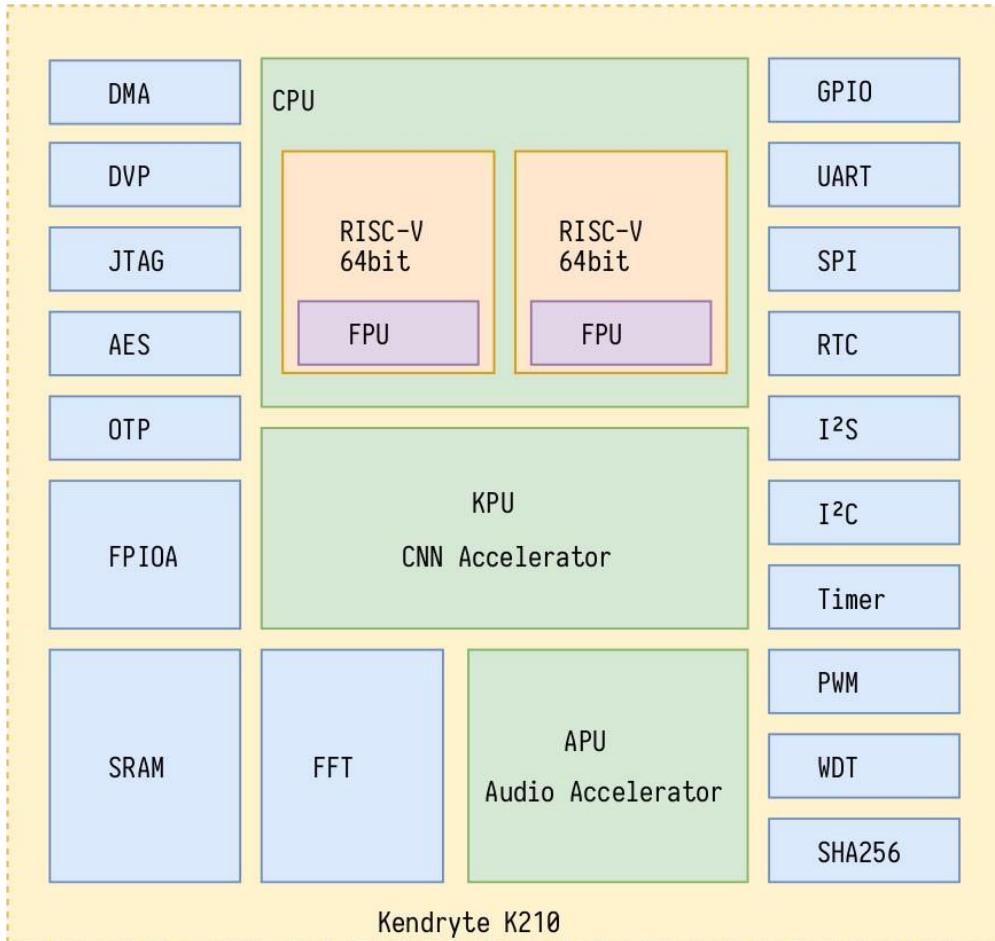
En esta sección abarcaremos los diferentes conceptos de interés en nuestro proyecto, desde la definición del concepto “*Internet of Things*” hasta la explicación del lenguaje *MycroPython* y el funcionamiento de los sistemas embebidos.

#### 3.3.1 Arquitectura RISC-V

En los últimos años hemos visto como el software libre y el código abierto está tomando cada vez más importancia en el mundo tecnológico con la creación de multitud de software, que, gracias a los diferentes usuarios es capaz de hacer competencia a las compañías privadas.

En el mundo del hardware libre este concepto está un poco más abandonado y es que es muy difícil encontrar microprocesadores libres, y los pocos que hay tienen una potencia muy limitada que no puede compararse con los privados.

RISC -V ha sido creado para hacer frente a este ámbito con la creación de una arquitectura de software libre basado en el diseño de tipo RISC. Con la creación de este conjunto de instrucciones cualquiera puede desarrollar su propio microprocesador, como es el caso del Kendrite K210, que es el microprocesador que está incluido en nuestra placa.



**Figura 29:** Estructura del procesador Kendrite K210

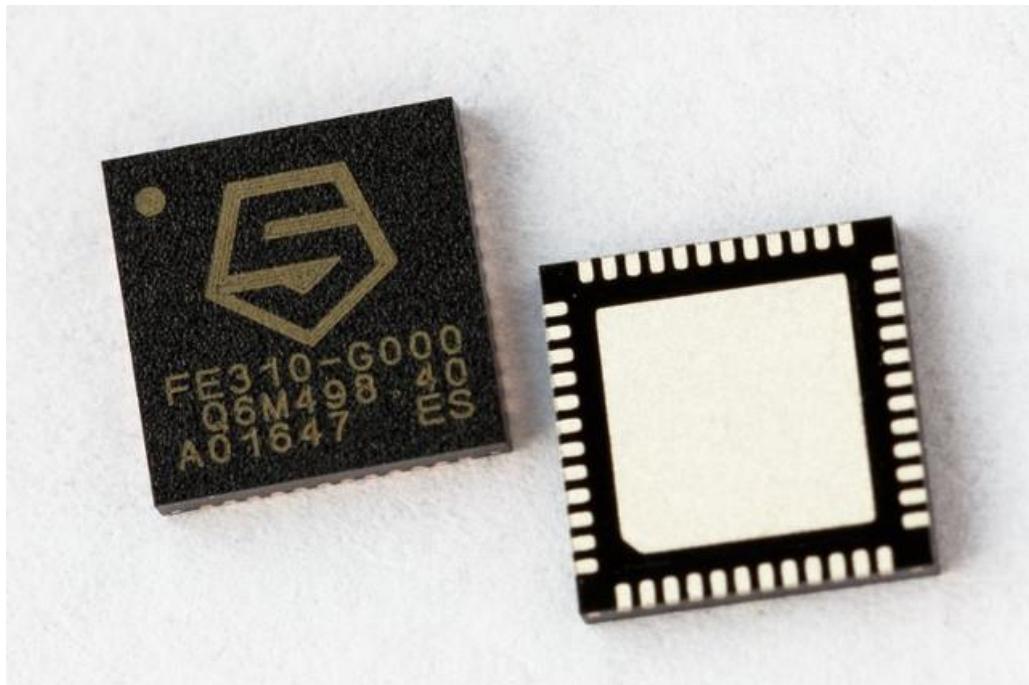
Desde que esta arquitectura empezó a dar sus primeros pasos en 2010 se han creado multitud de procesadores orientados a multitud de proyectos diferentes, incluidos los dispositivos móviles, donde parece que ha tenido gran repercusión gracias a su gran rendimiento y a la plataforma de 64 bits que dota a los dispositivos como Arduino o Raspberry de procesadores potentes por poco dinero y totalmente libres.

Uno de los cambios más importantes que ha introducido este nuevo concepto de arquitectura es el uso de ISA abierto con la misión de reducir la complejidad de los repertorios de instrucciones que actualmente dominan el mercado (Intel x86 y ARM).

El principal objetivo es conseguir un ISA reducido, estándar, modular y abierto que pueda tener extensiones para los diferentes ámbitos que quiera ser aplicado, evitando los sobrecostes por regalías y desarrollo.

RISC-V permite un cambio formativo tanto en las empresas como en el ámbito académico, permitiendo su estudio desde el inicio de esta, llegando a comprender el funcionamiento de la arquitectura al completo antes de que llegue a ser explotado por las grandes compañías, que ya están echando un ojo a este tipo de procesadores.

Es de importante mención la importancia de la inclusión de esta tecnología relativamente nueva en un proyecto de esta índole, ya que demostramos uno de sus usos a este nuevo tipo de hardware libre, que, si sigue por el buen camino, llegará a todos nuestros dispositivos electrónicos como los smartphones y darán un nuevo sentido a las placas controladoras, que serán capaces de desarrollar tareas más complejas.



**Figura 30:** Uno de los procesadores RISC-V por la marca [SiFive](#)

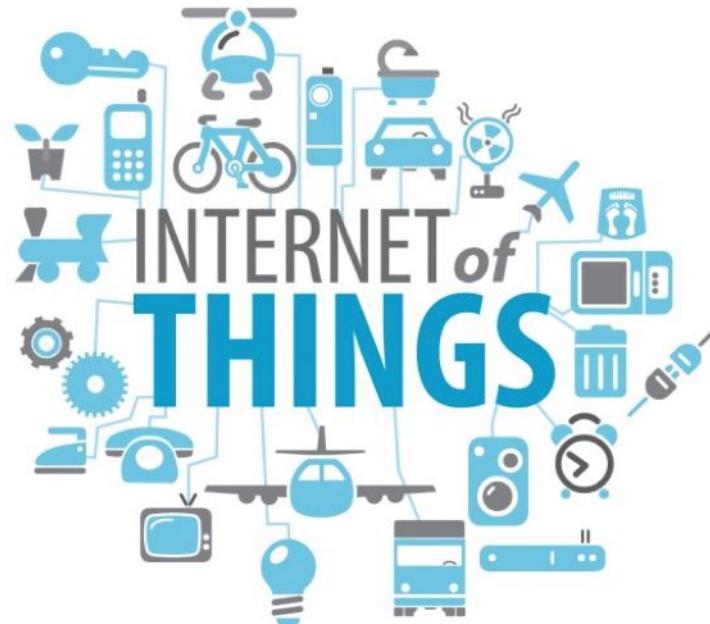
En la actualidad siguen surgiendo nuevos hitos relacionados con esta tecnología, generando nuevas [noticias](#) de investigación y desarrollo casi mensualmente. Como fichero de interés tenemos el primer volumen con el manual de esta estructura, creado en 2011 [[Andrew Waterman et al, 2011](#)].

### 3.3.2 Internet of Things (IoT)

El internet de las cosas (IoT) es la agrupación e interconexión de los diferentes dispositivos y objetos a través de algún medio, normalmente a través de la red, donde todos ellos pueden ser visibles e interactuar entre sí. Con el continuo crecimiento de la nube, el big data y las tecnologías móviles los objetos físicos pueden compartir y recopilar datos con una intervención humana mínima.

En nuestro caso, el poder crear un dispositivo capaz de reconocer cualquier imagen y clasificarla según un modelo CNN sin la necesidad de control de un humano, como por ejemplo el funcionamiento de una cámara de seguridad o el sistema de control de una cadena de montaje es una gran ventaja en el mundo actual donde está todo interconectado y automatizado.

Actualmente este concepto está tomando un gran auge con la incorporación de dispositivos de control en los diferentes electrodomésticos como las lavadoras y o los frigoríficos, que son capaces de comunicarte que ha acabado el ciclo de lavado o que incluso los yogures están a punto de caducar a través de una aplicación, la cual puedes instalar en tu *smartphone* o *tablet*, manteniéndote al tanto de todo lo que ocurre en tu casa con tan solo tener a mano tu teléfono.



**Figura 31:** Internet of Things

El concepto IoT está dado por hecho en nuestro día a día. Actualmente, es difícil no imaginarse un reloj inteligente que no se conecte con el dispositivo y que sea capaz de enviar correos o de responder llamadas.

Así mismo, el control de transportes, los horarios del metro o de autobús están todos controlados desde un dispositivo móvil, sabiendo a tiempo real el estado del transporte o lo que tarda en llegar el repartidor a tu casa, en el caso de las compañías de transporte.

La idea de construir un dispositivo compatible con la nueva tendencia de incluir “inteligencia” a cada objeto cotidiano nos acerca un poco más al mundo del futuro, consiguiendo que nuestro proyecto aún más relevancia

En el ámbito profesional las aplicaciones del IoT se puede encontrar fácilmente en los trabajos relacionados con la medicina, usando *weareables* que permitan monitorizar el estado y recuperación de un cierto paciente sin la necesidad de un control exhaustivo de un médico.

En los procesos industriales o en las máquinas expendedoras se podría usar como control de calidad, tanto de los productos que fabrica como de los productos que vende, pudiendo desactivar por ejemplo un determinado producto si detecta que presenta algún fallo.

Esto último podría fácilmente implementarse con un proyecto parecido a este, usando la visión por computador junto a una placa capaz de analizar cada producto vendido, detectando del estado del producto antes de venderlo.

### 3.3.3 TensorFlow Lite

---

[TensorFlow](#) es el *framework* creado por Google para el desarrollo de algoritmos relacionados con el *Deep Learning*. Este *framework* es el que usaremos para el desarrollo de nuestro modelo de clasificación en la primera parte del trabajo, sin embargo, el modelo a desarrollar debe estar implementado por [TensorFlow Lite](#), que se trata de una versión reducida de TensorFlow, la cual es la soportada para el posterior traspaso a nuestra placa.

Este tipo de *framework* reducido fue creado principalmente para su incorporación en dispositivos móviles que normalmente no tienen tanta capacidad de computo como los grandes ordenadores personales. Está especialmente diseñado para su uso junto microcontroladores en pequeños dispositivos, como nuestra placa MaixBit, la cual trabaja con su propio tipo de modelo, que son exportados directamente de los generados por la plataforma Tensorflow Lite.



## TensorFlow Lite

*Figura 32:* Logo TensorFlow Lite

La propia plataforma contiene multitud de ejemplos con las diferentes posibles aplicaciones de este tipo de *framework*. Entre ellos, destacan la clasificación de imágenes o la detección de objetos desde los diferentes dispositivos móviles.

Son los propios desarrolladores de TensorFlow los que recomiendan el entrenamiento de sus modelos desde el ordenador personal, y una vez entrenados, convertirlos a los modelos en formato reducido que pueden ser exportados a las plataformas.

Sobre la compatibilidad, la mayoría de las funciones de TensorFlow están disponibles en su versión Lite, excepto las relacionadas con *Float16* y *Strings*, ya que por ejemplo son las que más recursos necesitan. Las compatibilidades específicas vienen expresadas en la [página oficial](#) de la empresa.

En nuestro proyecto usaremos la versión completa de TensorFlow 2.0, el cual incorpora como última actualización Keras, una biblioteca dedicada a las redes neuronales que permite expresar de forma más sencilla y clara los pasos para su entrenamiento.

### 3.3.4 MicroPython

---

El lenguaje de programación usado para crear la estructura en TensorFlow 2.0 y [Keras](#) es [Python](#), un lenguaje de programación muy versátil el cual tiene varias bibliotecas dedicadas al aprendizaje profundo, facilitando de forma considerable la instalación de los diferentes recursos.

Como ya hemos mencionado anteriormente, estamos usando la herramienta [Anaconda](#) para desarrollar las diferentes pruebas del trabajo y la posterior creación del modelo.

Esta herramienta trabaja con Python clásico mientras que el software de la placa trabaja con una versión reducida de este, que está especialmente creado para su uso en dispositivos con menores recursos como los microcontroladores.



**Figura 33:** MicroPython

MicroPython es una implementación de Python 3 que incluye una versión reducida de sus diferentes librerías y que está optimizado para su uso en sistemas restrictivos y microcontroladores, como en nuestro caso.

La mayoría de las funciones presentes en el lenguaje original son soportadas por esta versión reducida para intentar que el traspaso de código desde el ordenador al microcontrolador sea lo más sencilla posible.

MicroPython es un compilador de Python completo y que es ejecutado en un [bare-metal server](#). Este tipo de lenguaje normalmente se ejecuta desde el *prompt* propio de la placa o dispositivo, ejecutando los comandos escritos inmediatamente después de su escritura o permitiendo la lectura de ficheros escritos de antemano directamente desde el sistema de archivos.

El software presente en nuestra placa contiene este *prompt* al que se puede acceder o bien desde el programa de control de la placa MaixBit IDE o bien desde programas como [PuTTY](#) una vez que ha sido flasheada la placa.

### 3.3.5 Sistemas embebidos

---

Los sistemas embebidos son unos dispositivos electrónicos diseñados para realizar unas funciones específicas en tiempo real, aunque no es una característica por defecto.

A diferencia de los ordenadores convencionales, que están diseñados para desarrollar una gran cantidad de tareas, estos sistemas están únicamente especializados en la realización y satisfacción de tareas muy específicas.

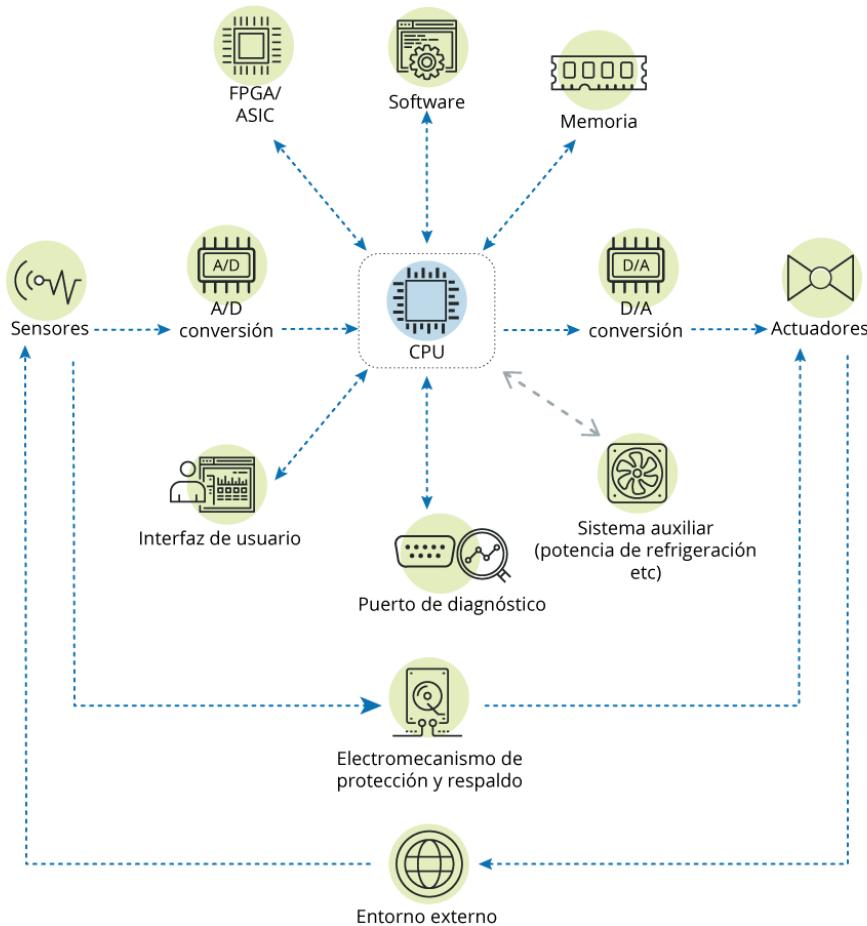
Aunque no es muy común hablar de este tipo de sistemas, estos están presentes en la mayoría de los dispositivos electrónicos conocidos, como una fotocopiadora o un ascensor, controlando su correcto funcionamiento y su seguridad.

Estos sistemas, aunque parezca que solo sirven de soporte para sistemas de mayor magnitud, son completamente funcionales y su principal objetivo es efectuar tareas de control.

Como podemos ver en la figura [34](#), los sistemas embebidos pueden estar compuestos por multitud de herramientas incluyendo incluso la interfaz de usuario, como es en el caso de nuestro trabajo.

La mayoría de las placas y microcontroladores se tratan de sistemas de este tipo, trabajando directamente desde su interfaz sin necesidad de un sistema operativo.

## COMPONENTES DE LOS SISTEMAS EMBEDIDOS

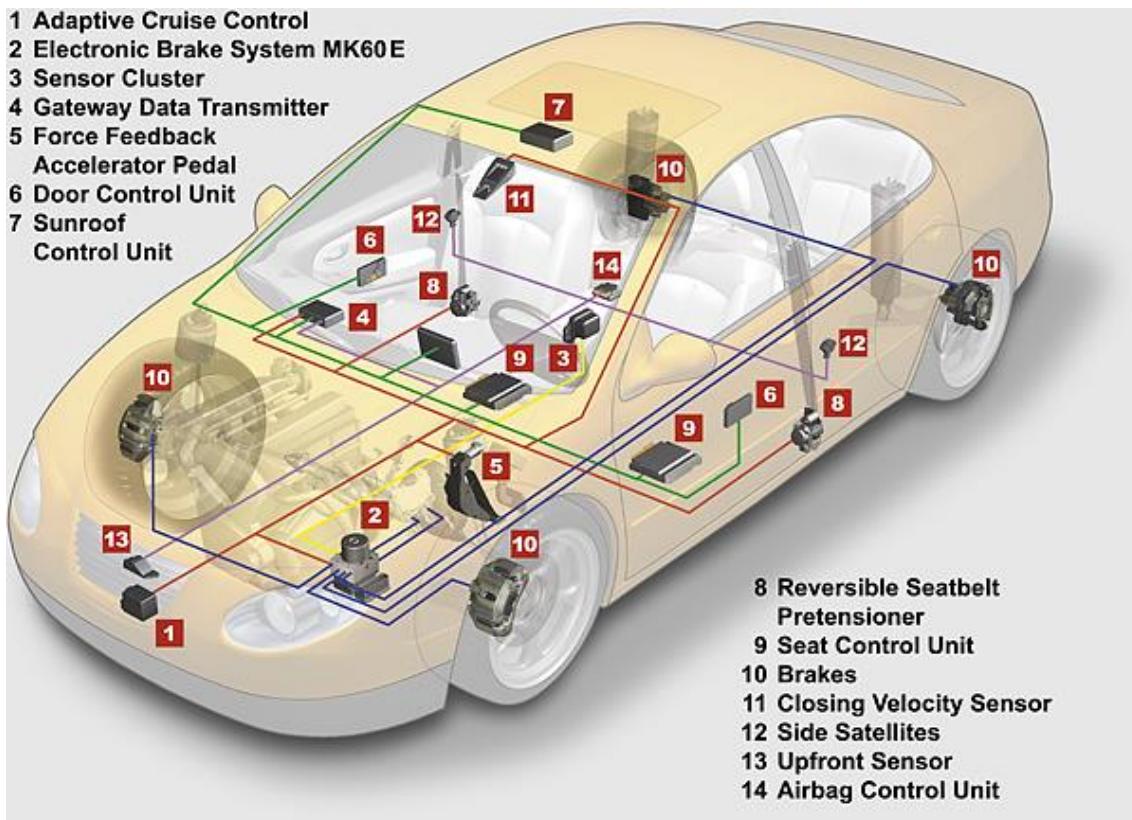


**Figura 34:** Componentes de los sistemas embebidos

Estos sistemas, al estar orientados a una tarea en concreto, realizan las operaciones a las que están especializados a gran velocidad, suponiendo una gran ventaja frente a los sistemas de propósito general. Por ejemplo, nuestra placa está orientada a la ejecución de sistemas de *Deep Learning*, teniendo unas características especiales que ayudan a realizar las operaciones necesarias en un menor tiempo que un sistema general.

La estructura de estos pequeños dispositivos está compuesta normalmente por un microcontrolador que incluye a su vez las interfaces de entrada junto a una interfaz externa para efectuar el monitoreo del estado y el diagnóstico del sistema.

La ventaja de que incluya únicamente los elementos necesarios para realizar su función es su bajo precio, y es que estos sistemas normalmente son producidos en grandes cantidades, siendo interesante el abaratamiento de estos.



**Figura 35:** Sistemas embebidos presentes en un coche

Otra ventaja de este tipo de sistemas es su bajo consumo y la poca necesidad de memoria física que necesitan para ser usados, aunque algunos sistemas no lo incluyen y es necesario sistemas de almacenamiento externos como es en nuestro caso, que necesitamos de una microSD para traspasarle el modelo CNN entrenado.

### 3.4 Diseño de un modelo portátil.

Teniendo en cuenta todos los conceptos vistos en capítulos anteriores y usando la estructura estudiada de redes convolucionales, vamos a comenzar con el diseño de nuestro modelo.

En este capítulo expondremos de forma detallada los pasos necesarios para construir este modelo, desde los requisitos previos para su construcción hasta su incorporación en la placa.

De forma general, la generación del modelo depende de los datos de entrada y los datos del algoritmo. Una vez generado analizamos los resultados con distintas fórmulas que nos indican como de “bueno” es.

Antes de comenzar el desarrollo de este modelo, debíamos estudiar las diferentes limitaciones expuestas por la placa, siendo importante limitar desde una primera instancia las capas usadas junto a los tamaños internos de la red. Comencemos pues con el estudio de estas características.

### 3.4.1 Limitaciones

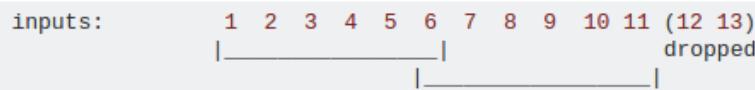
---

Uno de los principales factores para tener en cuenta es la capacidad máxima de memoria de la placa, siendo imposible incluir dentro de ella redes de un tamaño excesivo, teniendo en mucho de los casos en recortar el tamaño de las principales capas.

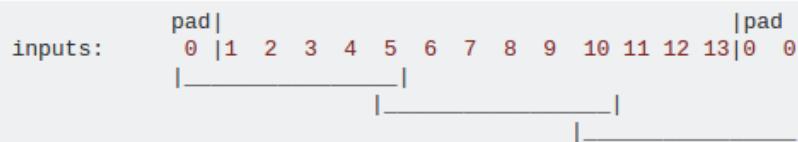
Otro factor importante es la capacidad de los modelos tipo “kmodel”, que son los únicos que pueden ser leídos con el software de la placa. Este tipo de modelos no incluyen la incorporación de las capas más modernas e incluyen un tipo específico de *padding*, teniendo que usar explícitamente este.

Con limitaciones tan sensibles como esta nos indica de una forma más o menos guiada la implementación de un modelo valido, intentando elaborar con capas básicas el modelo para nuestro problema.

- "VALID" = without padding:



- "SAME" = with zero padding:



**Figura 36.** Diferencia entre los tipos de padding

El factor de la memoria depende tanto de la capacidad de la placa como el uso que le demos a la memoria interna, y es que, si cargamos el modelo sin usar la MicroSD la capacidad de este se ve bastante reducida a la hora de ejecutarlo.

Nuestro principal objetivo es encontrar un modelo capaz de ser ejecutado por la placa, reduciendo al máximo el número de parámetros, pero que, a su vez, maximice los valores de precisión en la clasificación de las clases.

Partiendo de estos conceptos podemos comenzar con el estudio de las diferentes alternativas viables para elaborar un buen clasificador, siempre apoyándonos sobre el problema al que va dirigido.

### 3.4.2 Datos necesarios

---

Antes de comenzar con la implementación en sí del modelo debemos estudiar en profundidad nuestro problema, tanto para saber cómo enfocar el procedimiento como para buscar un buen *dataset* de datos sobre las clases.

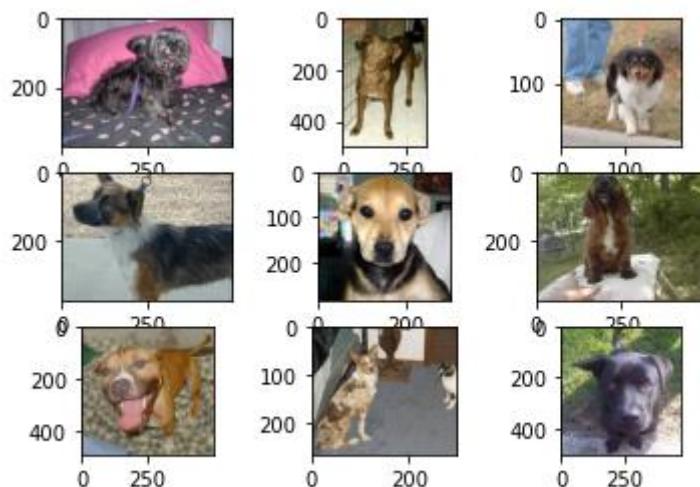
En nuestro caso, al tratarse de un problema conocido de clasificación como es la diferenciación de perros y gatos en imágenes, encontrar un *dataset* no ha sido ningún problema. El *dataset* usado en el trabajo ha sido obtenido de [Kaggle](#), el cual incluye un conjunto de datos de etiquetados y otro no etiquetado.

Al trabajar con imágenes debemos tener en cuenta varios factores que pueden suponer un problema para el entrenamiento: el ruido presente en las imágenes, problemas de etiquetado o incluso imágenes no válidas.

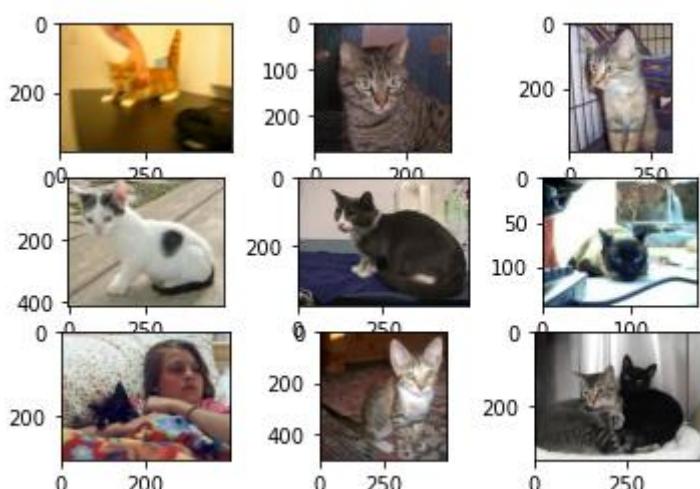
En este tipo de problemas es muy importante disponer de un gran número de imágenes de estudio; cuantos más ejemplos puedas darle a la red para que aprenda, mayor será el número de aciertos que tendrá posteriormente con imágenes desconocidas.

En nuestro caso, al tratarse de un problema de clasificación binaria, el número de imágenes no iba a suponer un problema, pero quizás si lo pudiera llegar a ser el tipo de imágenes de las que se trataban, ya que la mayoría tenían las mismas características.

### PERROS



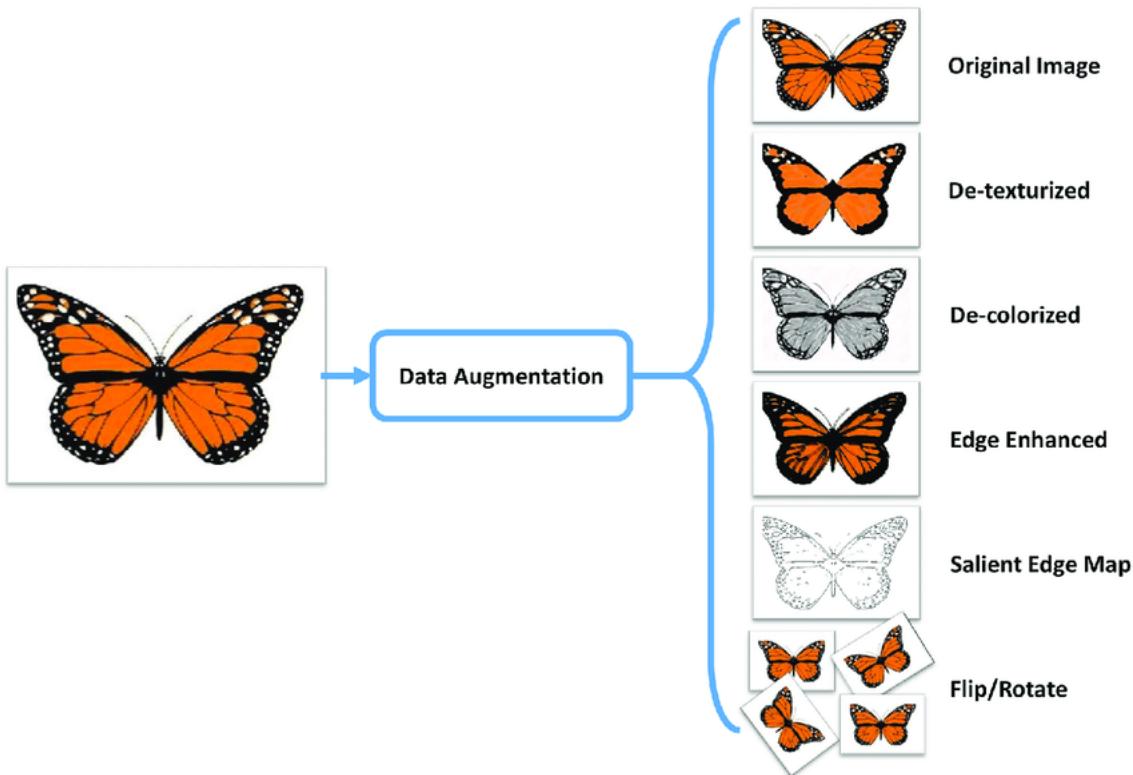
### GATOS



*Figura 37. Imágenes etiquetadas del dataset*

Debido a las innumerables situaciones que pueden darse en las imágenes hemos decidido aplicar técnicas de tratamiento de imágenes para aumentar el número de ejemplos diferentes.

Para conseguir un *dataset* más completo y variado se aplica la técnica de *data augmentation*, la cual consiste en aplicar diferentes variaciones a las imágenes con el fin obtener un modelo invariante a las posibles situaciones venideras.



**Figura 38. Ejemplo de Data Augmentation**

Otro proceso que debemos llevar a cabo antes de generar la estructura del modelo es la clasificación de las imágenes en diferentes categorías que nos ayuden a construir un modelo de calidad.

Normalmente, el conjunto entero de *dataset* se divide en tres grupos:

- **Imágenes de entrenamiento:** Estas imágenes se usan para obtener características y actualizar los parámetros internos de la red.
- **Imágenes de validación:** Sirven para comprobar el rendimiento del modelo que se va construyendo durante el entrenamiento.
- **Imágenes de testeo:** Estas imágenes no se usan durante el proceso de entrenamiento, se usan una vez generado el modelo completo para comprobar el rendimiento de la red sobre imágenes nunca vistas.

Es importante tener bien separados cada uno de los conjuntos por su función, ya que, si usáramos las mismas imágenes de entrenamiento para testeo podrían producirse valores de inexactitud en los resultados de precisión del modelo.

Con las imágenes etiquetadas de nuestro *dataset* conseguimos las imágenes necesarias para las dos primeras categorías, dejando un 25% del total para validación y el resto para entrenamiento.

Las imágenes sin etiquetar tuvimos que tratarlas manualmente para conseguir separarlas en clases. De este grupo clasificamos el 50% del total de estas imágenes y las transformamos en el conjunto de testeo.

La división en estas tres categorías nos ayuda a llevar de una manera más sencilla el control de las imágenes usadas a la par de un control durante el entrenamiento, evaluando tanto la precisión como la perdida de los conjuntos de validación y entrenamiento.

Antes de comenzar el entrenamiento es importante dedicarle un tiempo al estudio de los datos que vamos a usar, comprendiendo las necesidades que tienen y buscando la mejor forma de trabajar con ellos.

### 3.4.3 Estructura de la red

---

Una vez clasificadas de forma correcta las imágenes de nuestro *dataset* debemos encontrar la mejor estructura de red que nos permita generar un modelo que cumpla nuestras condiciones.

La buena construcción de la red no depende solo de la cantidad de capas que contenga, también depende de cómo se configure cada una de ellas. Uno de los factores configurables que tienen más importancia en este proceso son la inicialización de los valores internos de la red.

Las primeras ideas que puedes tener para que se produzca esta inicialización puede ser usar ceros o hacerlo de forma aleatoria, siendo unas opciones muy poco eficientes y que perjudicarían en gran medida al sistema.

Si los valores iniciales son muy bajos, los datos de entrada que van pasando por las capas son demasiado pequeños como para ser útiles para el aprendizaje; si por el contrario son muy altos, los datos son demasiado grandes como para aprender de ellos.

Para encontrar unos datos iniciales óptimos para el entrenamiento, podemos definir dos tipos de inicializaciones:

- **Xavier/Glorot Initialization:** Este tipo inicializa los pesos de la red dibujando una distribución con media 0 y una varianza específica.

$$Var(w_i) = \frac{1}{fan\_in}$$

donde *fan\_in* representa el número de neuronas entrantes.

Esta técnica dibuja ejemplos desde una distribución centrada en 0 con:

$$stddev = \sqrt{\frac{1}{fan\_in}}$$

donde *fan\_in* es el número de unidades de entrada en el tensor de pesos.

- **He Normal (He-et-al) Initialization:** Este método de inicialización se hizo famoso con la publicación del *paper* por parte de He-et-al en 2015 [[He et al, 2015](#)]. Se trata de un modelo similar al de Xavier, con la particularidad de que los pesos son inicializados dependiendo del tamaño de la capa anterior de neuronas. Usando activación RELU:

$$Y = w_1x_1 + w_2x_2 + \dots + w_nx_n$$

$$Var(w_i) = \frac{2}{fan\_in}$$

Esta técnica ayuda a obtener una inicialización no tan aleatoria como la anterior, consiguiendo un descenso por gradiente más rápido y eficiente. Distribuye las muestras de la misma forma que la inicialización anterior, pero usando la siguiente desviación:

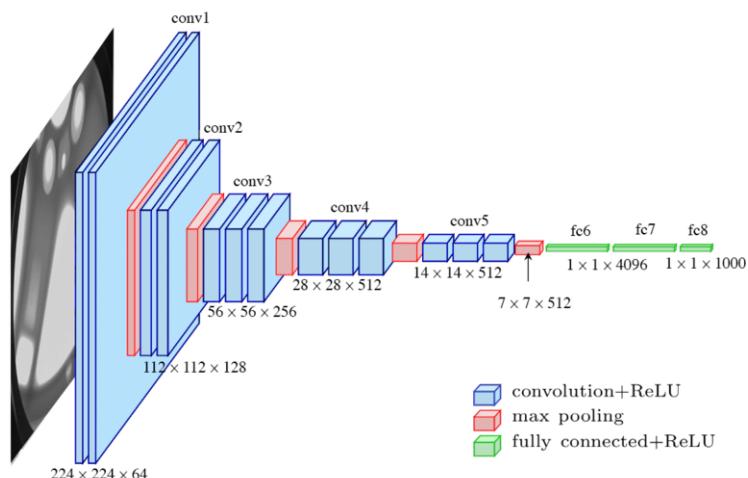
$$stddev = \sqrt{\frac{2}{fan\_in}}$$

donde *fan\_in* es el número de entradas en el tensor de pesos.

Estos dos métodos son dos ejemplos de los modelos de inicialización que se usan en el entrenamiento de redes neuronales convolucionales, demostrando grandes resultados en problemas conocidos como [ImageNet](#).

Conociendo diferentes maneras de inicializar las capas de nuestra red podemos pasar a estudiar qué tipo de estructuras pueden tener nuestras redes.

Como nuestro problema requiere de una red sencilla vamos a presentar una de las estructuras más usadas para este tipo de problemas de clasificación binaria, como puede ser la arquitectura VGG, presentada por primera vez en el *paper* de Karen Simonyan [[Simonyan y Zisserman, 2014](#)].

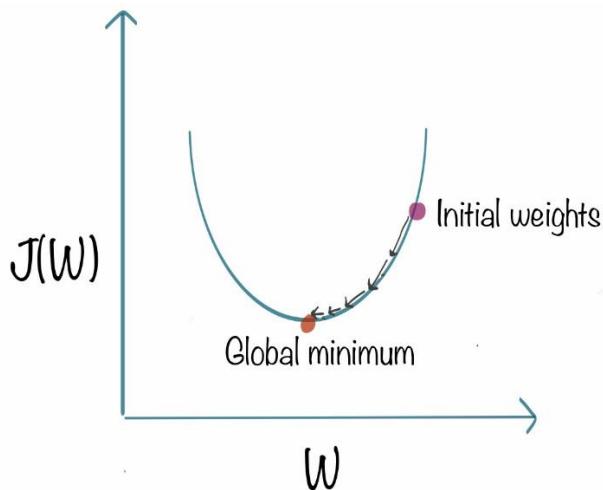


**Figura 39.** Estructura de Red VGG 16

Este tipo de red nos puede servir como punto base para la construcción de nuestra red. Al tratarse de un problema conocido podemos encontrar diferentes [páginas](#) donde se ha resuelto este problema con el uso de esta estructura, garantizándonos buenos resultados.

Con la inicialización y la estructura planteadas nos haría falta conocer una función de optimización que nos ayude a conseguir buenos resultados en poco tiempo. Al igual que con los inicializadores, vamos a ver dos ejemplos de los más populares.

- **Stochastic Gradient Descent** o descenso por gradiente estocástico (SGD). Se trata de un método de optimización basado en el descenso por gradiente, el cual trata de actualizar cada uno de los pesos de la red basándose en el error cometido en la capa de salida.



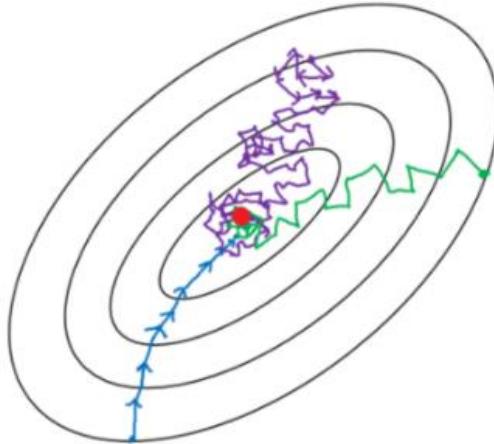
**Figura 40.** Descenso por gradiente

Uno de los principales problemas de este método es la necesidad de computar todos y cada uno de los pesos de la red para que se produzca la actualización, volviéndose completamente ineficiente para problemas de gran tamaño.

El descenso por gradiente estocástico, basado en el documento de Kiefer de 1952 [[Kiefer y Wolfowitz, 1952](#)], trata de reducir al mínimo esta computación eligiendo de forma aleatoria un dato del conjunto total en cada iteración, cambiando únicamente ese valor.

Generalmente se usa un pequeño conjunto de valores “mini-batch” los cuales serán tratados en ese paso, tratando de mantener un punto medio entre la precisión proporcionada por el algoritmo descenso por gradiente y la velocidad dada por el algoritmo SGD.

- Batch gradient descent (batch size = n)
- Mini-batch gradient Descent (1 < batch size < n)
- Stochastic gradient descent (batch size = 1)



**Figura 41.** Diferencia entre los diferentes descensos por gradiente

- **Adam.** Este tipo de algoritmo de optimización es una extensión del algoritmo descenso por gradiente estocástico presentado por Diederik P. Kingma y Jimmy Ba en su *paper* “*Adam: A Method for Stochastic Optimization*” [[Kingma y Ba, 2017](#)].

Según los autores, Adam se basa en dos subclases del algoritmo SGD, en específico de AdaGrad y RMSProp, quedándose con los beneficios de ambos. Adam trata de adaptar el parámetro de aprendizaje basado en la media del primer y el segundo momento en los gradientes.

*For each Parameter  $w^j$*

*( $j$  subscript dropped for clarity)*

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta$  : Initial Learning rate

$g_t$  : Gradient at time  $t$  along  $\omega^j$

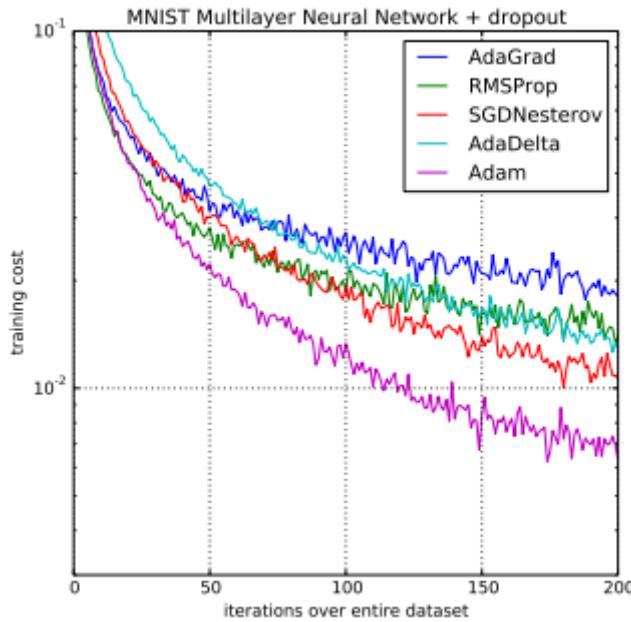
$\nu_t$  : Exponential Average of gradients along  $\omega_j$

$s_t$  : Exponential Average of squares of gradients along  $\omega_j$

$\beta_1, \beta_2$  : Hyperparameters

**Figura 42.** Algoritmo de optimización Adam

Como podemos ver en la figura 42 la actualización de los pesos se basa en las medias de los gradientes junto al gradiente en un tiempo en concreto, actualizado de forma controlada el *learning rate* y consiguiendo mejores resultados que otros algoritmos.



**Figura 43.** Comparación algoritmos de optimización (coste entrenamiento)

#### 3.4.4 Estudio de resultados

Una vez vistos todos los pasos necesarios para construir el modelo completo, debemos saber cómo de bueno es, estudiando datos como la precisión o la pérdida del modelo con respecto a las imágenes de nuestro *dataset*.

Como bien sabemos, durante el entrenamiento de la red se aprenden las características a partir de las imágenes de entrenamiento, mientras que, al final de cada época, se usan las imágenes de validación para comprobar cómo se comporta frente a imágenes “nunca vistas”.

Al final de cada época obtenemos varios datos de interés para valorar la evolución en el proceso de aprendizaje de nuestro algoritmo, siendo posible actuar en función de ellos o incluso programar una salida si esos valores alcanzan un cierto valor.

```
Epoch 1/100
293/293 [=====] - 266s 909ms/step - loss: 0.6799 - accuracy: 0.5693 - val_loss: 0.6528 - val_accuracy: 0.6023
Epoch 2/100
293/293 [=====] - 156s 532ms/step - loss: 0.6374 - accuracy: 0.6368 - val_loss: 0.6003 - val_accuracy: 0.6762
Epoch 3/100
293/293 [=====] - 156s 531ms/step - loss: 0.5980 - accuracy: 0.6798 - val_loss: 0.5550 - val_accuracy: 0.7203
Epoch 4/100
293/293 [=====] - 161s 549ms/step - loss: 0.5633 - accuracy: 0.7123 - val_loss: 0.6485 - val_accuracy: 0.6524
Epoch 5/100
293/293 [=====] - 160s 547ms/step - loss: 0.5301 - accuracy: 0.7421 - val_loss: 0.4824 - val_accuracy: 0.7687
```

**Figura 44.** Datos obtenidos en cada una de las etapas

Al final del entrenamiento, para comprobar la ciencia cierta la efectividad de nuestro modelo debemos medir otros parámetros tales como la sensibilidad o la especificidad con el tercer conjunto de imágenes, las cuales no han sido usadas en ningún momento durante el aprendizaje.

Para elaborar de forma correcta los resultados debemos usar la matriz de confusión de nuestro problema, la cual incluye información como la cantidad de imágenes que han sido acertadas de cada clase y las que no, indica con cuál de ellas se ha equivocado.

	airplane	4	21	8	4	1	5	5	23	6
True Class	923	5	972	2				1	5	15
airplane	5	972	2					1	5	15
automobile										
bird	26	2	892	30	13	8	17	5	4	3
cat	12	4	32	826	24	48	30	12	5	7
deer	5	1	28	24	898	13	14	14	2	1
dog	7	2	28	111	18	801	13	17		3
frog	5		16	27	3	4	943	1	1	
horse	9	1	14	13	22	17	3	915	2	4
ship	37	10	4	4		1	2	1	931	10
truck	20	39	3	3			2	1	9	923
Predicted Class	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck

*Figura 45. Ejemplo de Matriz de Confusión*

En esta matriz podemos ver por filas las clases verdaderas y por columnas las clases predichas, de tal forma que, si estuviéramos en un mundo ideal, solo habría valores en la diagonal.

En la primera fila de la figura 45 podemos ver la predicción de la clase avión, que a primera vista tiene muy buena precisión ya que el valor correspondiente con su columna tiene un valor elevado comparado con los valores en el resto de la fila. Estos otros valores indican que clase predijo el modelo, teniendo una visión más clara de los posibles fallos que puede cometer nuestro sistema.

A partir de esta, podemos obtener de forma sencilla un estudio sobre la efectividad de nuestro modelo en cada una de las clases. El estudio de las filas, las columnas y diagonales nos pueden dar muchísima información importante.

		Predicted							
		False Positives							
		TN							
Actual		True Negatives	0	0	0	0	0	0	
1	0	False Negatives	2	3	0	0	0	0	TP
0	3	1	1	0	0	0	1	FP	FN
1	1	0	3	0	0	0	5	0	TN

**Figura 46.** Valores obtenibles a partir de la matriz de confusión

Vamos a ver algunos de los valores que podemos estudiar para ayudarnos a medir esta efectividad en función de los valores TP (*True Positive*), TN (*True Negative*), FP (*False Positive*) y FN (*False Negative*) obtenidos a partir de la matriz:

- **Precisión:** La precisión o *accuracy* nos indica cuantas imágenes ha clasificado correctamente con respecto al total.

$$ACC = \frac{TP + TN}{TP + FP + TN + FN}$$

- **Positive Predictive Value:** Este valor nos indica la cantidad de positivos acertados con respecto al total de los ejemplos positivos predichos, tanto los verdaderos como los falsos.

$$PPV = \frac{TP}{TP + FP}$$

- **Negative Predictive Value:** Este valor es el contrario al anterior y nos indica la cantidad de ejemplos negativos han sido clasificados correctamente con respecto a todos los negativos predichos.

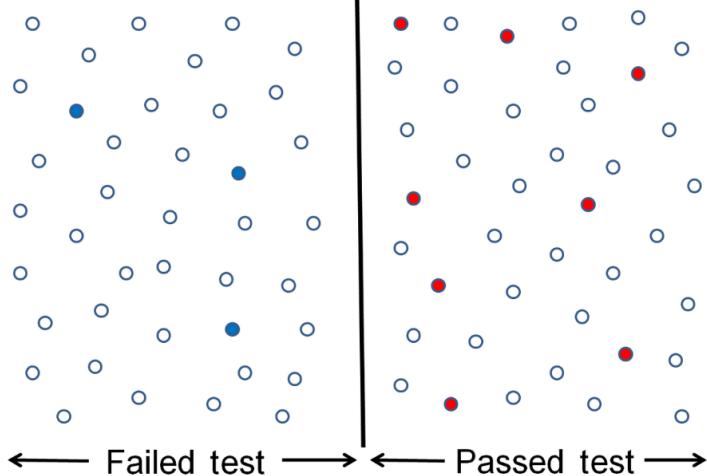
$$NPV = \frac{TN}{TN + FN}$$

- **Sensitivity:** La Sensibilidad del modelo da un valor indica la probabilidad de que un valor dado como positivo sea en realidad positivo. Esto se realiza midiendo la cantidad de positivos verdaderos entre el total de positivos verdaderos y falsos negativos.

$$Sensitivity = \frac{TP}{TP + FN}$$

### High sensitivity      Low specificity

Few false negatives (blue)      Many false positives (red)



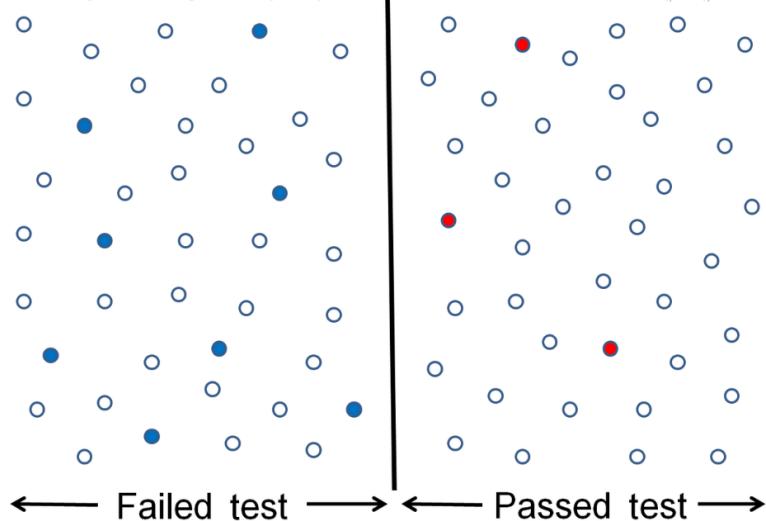
*Figura 47. Diferencia entre sensibilidad alta o baja*

- **Specificity:** La especificidad mide justo lo contrario que la sensibilidad, dando la probabilidad de que un valor dado como negativo sea en realidad un negativo real.

$$Specificity = \frac{TN}{TN + FP}$$

### Low sensitivity      High specificity

Many false negatives (blue)      Few false positives (red)



*Figura 48. Diferencia entre especificidad alta y baja*

- **False Positive Rate:** Este valor también conocido como ratio de falsa alarma indica la cantidad de falsos positivos con respecto a todos los ejemplos negativos, que son la unión de los falsos positivos y los verdaderos negativos. Cuanto más bajo, mejor es el modelo.

$$FPR = \frac{FP}{FP + TN}$$

- **False Negative Rate:** La ratio de falsos negativos nos indica lo mismo que el parámetro anterior, pero esta vez mirando los falsos negativos con respecto a los falsos negativos y verdaderos positivos. Al igual que el anterior, cuanto menor es el valor mejor es el modelo.

$$FNR = \frac{FN}{TP + FN}$$

- **False Discovery Rate:** Este valor nos indica la cantidad de falsos positivos que hay con respecto a todos los ejemplos que han dado positivo, dando un valor bajo si la mayoría de los ejemplos son verdaderos positivos.

$$FDR = \frac{FP}{TP + FP}$$

Teniendo en cuenta todos estos valores numéricos podemos obtener de forma precisa una conclusión sólida de la eficacia de nuestro modelo, sirviéndonos en mucho de los casos a solucionar posibles problemas que pudieran suceder en el entrenamiento.

Un ejemplo muy bueno es si la especificidad o la sensibilidad no tienen valores similares, dejando ver una inestabilidad entre la predicción de los ejemplos negativos frente a los positivos o viceversa, dando un posible punto a mejorar dentro del modelo.

Como hemos comentado a lo largo de la memoria, la evaluación propia del modelo es la que más información nos puede dar y sin duda la que mejor va a indicar la calidad de nuestro algoritmo, dándole validez y credibilidad.

### 3.4.5 Migración a la placa

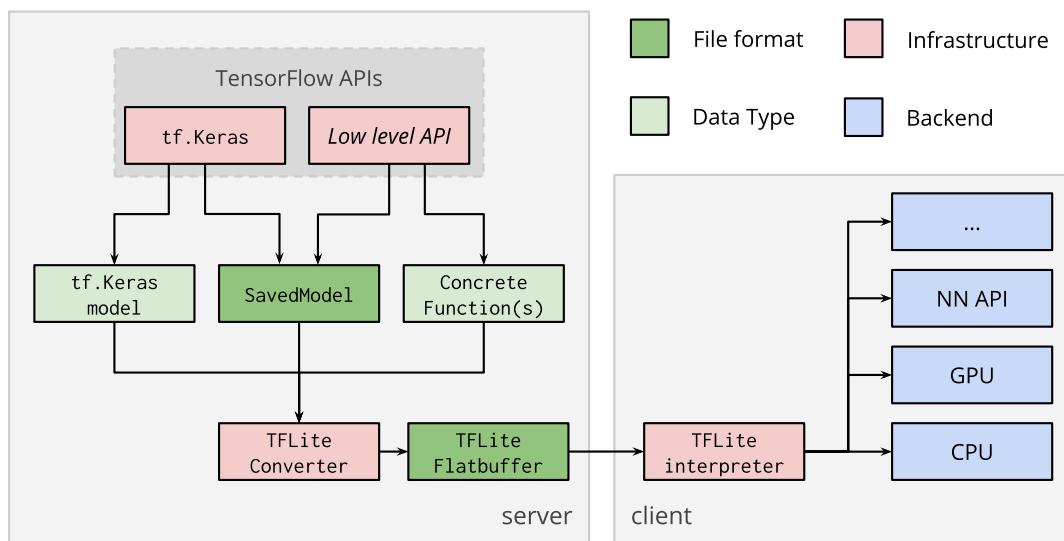
Tras conseguir un modelo valido debemos dar el siguiente paso dentro de nuestro trabajo, que es traspasar este a un formato valido para que pueda ser ejecutado por nuestra placa

El modelo anteriormente generado tiene como extensión “.h5”, extensión propia de los modelos de *Keras* en *TensorFlow*, siendo modelos completos y con multitud de datos en su interior.

El primer paso fundamental antes de comenzar a trabajar con los diferentes recursos que nos da la empresa creadora del dispositivo es reducir el tamaño del modelo original quedándonos únicamente con los datos de interés del modelo.

*TensorFlow* incluye un *framework* especializado en solucionar este tipo de problemas llamado *TensorFlow Lite*, el cual cuenta con una función específica capaz de convertir un modelo previamente entrenado en la versión original, a la versión reducida.

Este *framework* se especializa en la incorporación de modelos en dispositivos portátiles, tales como teléfonos móviles o dispositivos electrónicos (IoT), o incluso en servidores, usándose por ejemplo para el reconocimiento de voz en línea con Google Home.



**Figura 49.** Implementación en Dispositivos de TFLite

Una vez conseguimos un modelo en formato “.tflite” podemos empezar a tratar con el software específico de la placa, el cual se encuentra en formato de código en su [página oficial de GitHub](#).

Para poder usar todos los recursos debemos usar el sistema operativo Linux, sin embargo, haciendo unos pequeños cambios en los archivos, podemos conseguir ejecutarlos desde el terminar de [Ubuntu \(disponible en la Microsoft Store\)](#) sin ningún tipo de problemas.

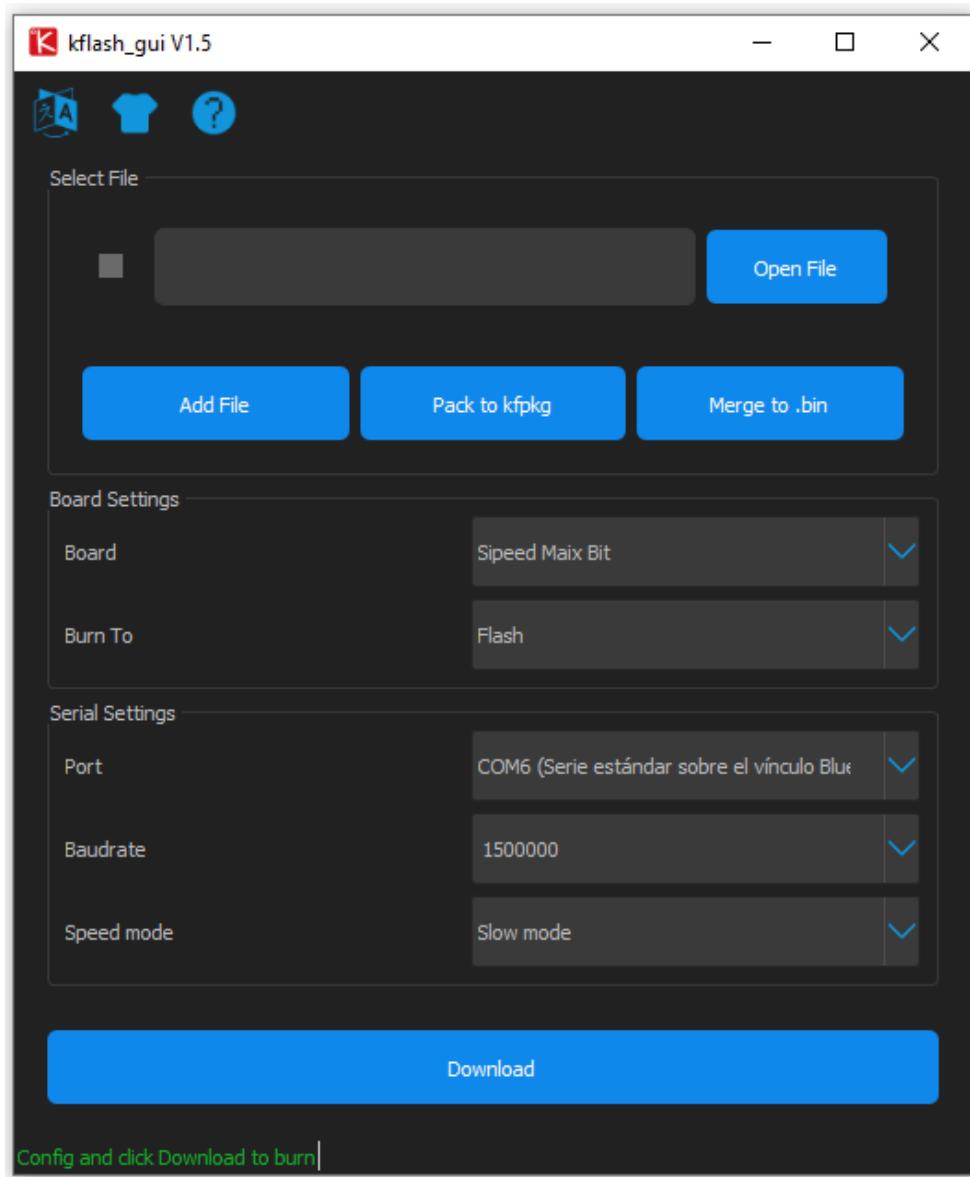
Como hemos mencionado en capítulos anteriores, el modelo que debemos conseguir tiene la extensión “.kmodel”, el cual se puede conseguir usando las herramientas expuestas en el [repositorio oficial](#) de la placa. Este repositorio viene explícitamente dado para su uso en Linux, teniendo que realizar ciertos cambios para su uso en Windows.

Para evitar problemas en este paso en futuras implementaciones, he creado mi propio [entorno de trabajo](#) donde se pueden usar directamente los ficheros de instrucciones “.sh” desde la terminar de Ubuntu en Windows 10 sin ningún tipo de problema.

Una vez terminada la conversión completa de nuestro modelo al formato valido únicamente nos queda realizar la incorporación en la placa, primero inicializando la placa con un firmware y posteriormente corriendo el modelo en su interior.

La placa, una vez conectada al ordenador puede ser cargada con diferentes versiones de firmware, siendo recomendable usar el que menos memoria ocupe, para conseguir más capacidad en nuestro modelo.

Este firmware se carga desde la aplicación K-flash, la cual es capaz de cargar tanto el *firmware* específico de la empresa como alguno personalizado por nosotros donde, se encuentre el modelo previamente cargado en su interior, consiguiendo una ejecución directa desde su interior.



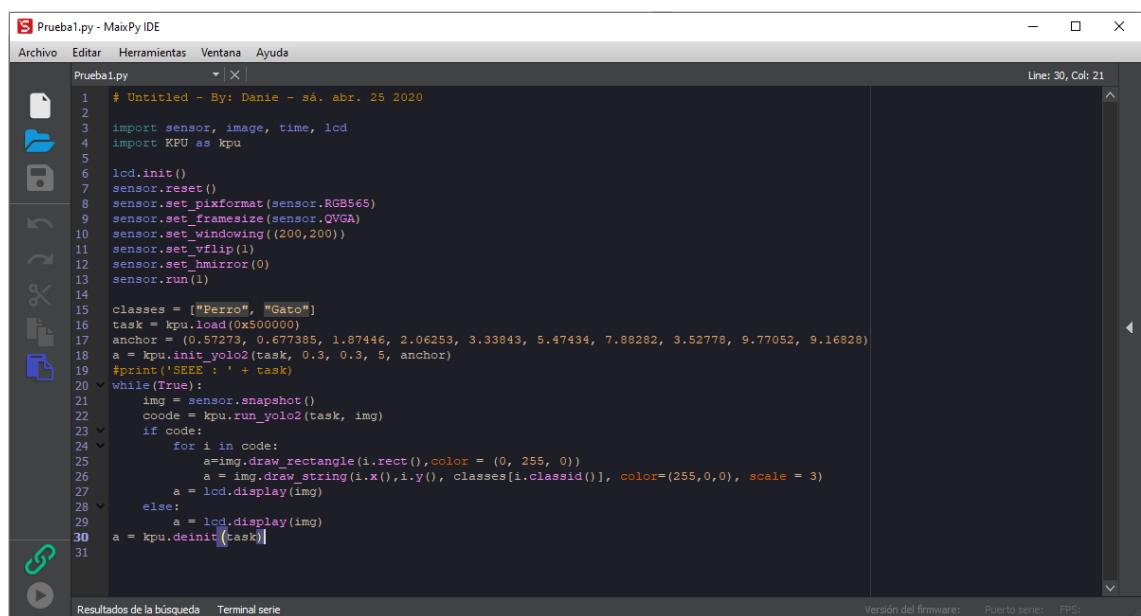
*Figura 50. Interfaz de la aplicación K-Flash*

Una vez la placa ha sido cargada con el *firmware* necesario podemos proceder con la ejecución del modelo en el interior de ella, haciendo uso del lenguaje de programación MicroPython, el cual es el usado por la placa.

Para conseguir comunicarnos con la placa nos encontramos con dos opciones, una específica para usarse con el firmware completo y otra pensada para usarse con el firmware mínimo:

- **Firmware completo:** Si podemos permitirnos cargar el firmware completo sobre nuestra placa podemos usar la interfaz gráfica que nos proporciona Sipeed llamada [MaixPy IDE](#).

Este programa incluye un editor de textos que se comunica directamente con la placa y es capaz de ejecutar una lista de instrucciones sobre la placa con tan solo pulsar un botón.



```

Prueba1.py - MaixPy IDE
Archivo Editar Herramientas Ventana Ayuda
Prueba1.py Line: 30, Col: 21
1 # Untitled - By: Danie - sá. abr. 25 2020
2
3 import sensor, image, time, lcd
4 import KPU as kpu
5
6 lcd.init()
7 sensor.reset()
8 sensor.set_pixformat(sensor.RGB565)
9 sensor.set_framesize(sensor.QVGA)
10 sensor.set_windowing((200,200))
11 sensor.set_vflip(1)
12 sensor.set_hmirror(0)
13 sensor.run(1)
14
15 classes = ["Perro", "Gato"]
16 task = kpu.load(0x500000)
17 anchor = (0.57273, 0.677385, 1.87446, 2.06253, 3.33843, 5.47434, 7.88282, 3.52778, 9.77052, 9.16828)
18 a = kpu.init_yolo2(task, 0.3, 0.3, 5, anchor)
19 #print('SEEK : ' + task)
20 while(True):
21     img = sensor.snapshot()
22     coode = kpu.run_yolo2(task, img)
23     if coode:
24         for i in coode:
25             a=img.draw_rectangle(i.rect(),color = (0, 255, 0))
26             a = img.draw_string(i.x(),i.y(), classes[i.classid()], color=(255,0,0), scale = 3)
27             a = lcd.display(img)
28     else:
29         a = lcd.display(img)
30 a = kpu.deinit(task)
31

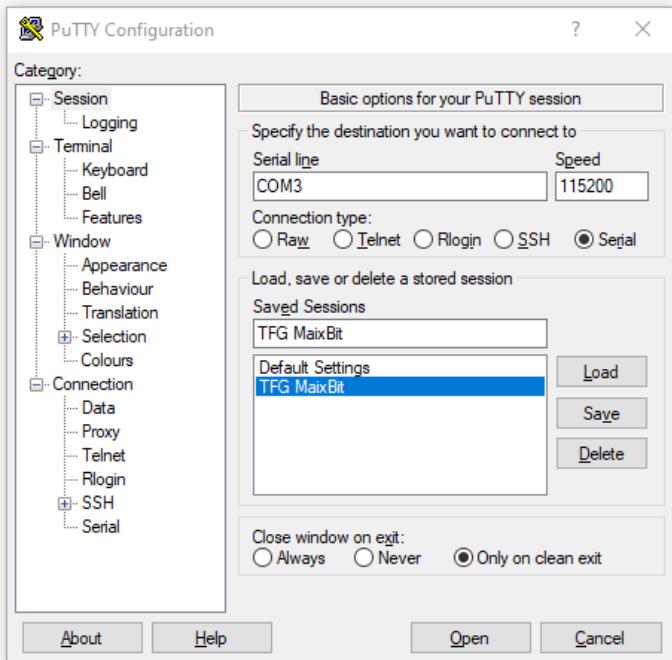
```

Resultados de la búsqueda Terminal serie Versión del firmware: Puerto serie: FPS:

*Figura 51. MaixPy IDE*

Usando este tipo de programa solo podemos cargar modelos muy pequeños debido a que gran parte de la memoria interior de la placa está usada para soportar este tipo de programas, quedando como alternativa la siguiente opción.

- **Firmware mínimo:** Usando el firmware mínimo para cargar el modelo debemos establecer la conexión con la placa usando un programa externo. En nuestro caso hemos usado [Putty](#), el cual nos permite abrir una ventana de comandos donde comunicarnos con la placa y escribir las instrucciones que queramos que se ejecuten.



**Figura 52.** Interfaz Putty

Una vez establecida la conexión podemos trabajar directamente desde la ventana de comandos, teniendo que usar los mismos comandos que con la opción anterior.

La mejor forma para comprobar si la placa funciona es primero ejecutarla usando la primera opción, inicializando los sensores y el LCD, comprobando que todo este correcto y elaborando un pequeño script donde se reinicie todo según queramos.

Una vez obtenido este código, podemos cargar el *firmware* mínimo, pudiendo copiar el código generado anteriormente directamente en la placa, ahorrándonos escribir una a una todas las instrucciones necesarias para su funcionamiento.

Tendiendo el firmware cargado podemos usar las librerías internas de la placa para cargar nuestro modelo y hacerlo funcionar analizando cada una de las imágenes que reciba. Este proceso se repetirá en un bucle infinito en el cual se tomará una imagen, se pasará por el modelo y posteriormente se mostrará en la pantalla indicando a la clase que más se asemeja.

La librería principal para el tratamiento de modelos es la denominada [KPU](#), la cual incluye varias funciones para trabajar con reconocimiento de formas con yolo2 y varias funciones para trabajar con clasificadores, como es nuestro caso.

---

# Capítulo 4

## Trabajo de experimentación

---

Una vez vista toda la teoría que engloba nuestro trabajo vamos a proceder con la explicación de los pasos seguidos en la parte práctica de nuestro trabajo, el cual se asemeja con lo expuesto en el final del capítulo anterior.

En este capítulo explicaremos tanto los diferentes pasos que hemos ido siguiendo hasta conseguir ejecutar el modelo en nuestra placa, hasta los diferentes datos y funciones que hemos ido usando para conseguir nuestro objetivo.

Para comprender de mejor manera la metodología usada y los diferentes pasos que hemos ido realizando vamos a dejar subido un repositorio de GitHub donde se encontrarán todos los recursos usados en la elaboración de este trabajo.

Como hemos incluido el [Anexo II](#) donde explicamos de forma resumida la manera de usar el repositorio para conseguir que el modelo funcione, en esta sección nos centraremos en justificar los pasos tomados y explicar de forma detallada los pasos y resultados obtenidos.

### 4.1 Objetivos y materiales

---

Nuestro principal objetivo es demostrar la creciente capacidad de procesamiento que están teniendo los diferentes dispositivos electrónicos en la actualidad, cada vez siéndonos más fácil realizar las diferentes tareas, tanto cotidianas como profesionales.

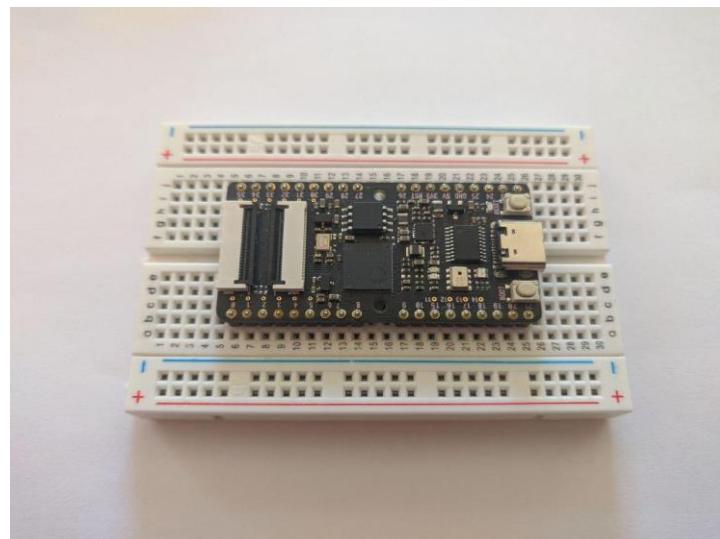
La incorporación de una red tan potente como son las convolucionales en un dispositivo tan pequeño y manejable nos abre un abanico de posibilidades a la hora de crear nuevas herramientas o mejorar alguna de las ya creadas.

Existen multitud de proyectos que tratan con este tipo de redes, algunos de ellos son desarrollados por grandes cadenas centradas en la automatización de cadenas de montaje y control de calidad y otras, en cambio, se dedican a proyectos a menor escala demostrando la efectividad de esta tecnología. Un proyecto referente de esto último es el realizado por Umut Ozkaya, el cual creo una red capaz de diferenciar la clase de basura en sus diferentes tipos con intención de crear un dispositivo capaz de reciclar automáticamente [\[Ozkaya y Seyfi, 2019\]](#).

Con la creación de nuestra pequeña red y su incorporación en el dispositivo daremos por demostrada la efectividad y eficiencia que podemos conseguir con herramientas accesibles para todos, suponiendo una nueva vía de aprendizaje para nuevos usuarios e incluso una herramienta muy versátil para usuarios experimentados, los cuales quieran aprender una nueva forma de usar el aprendizaje profundo.

Para conseguir elaborar un buen modelo y prototipo de clasificador de imágenes portátil hemos tenido que utilizar los siguientes materiales:

- **Imágenes de perros y gatos.** Estas imágenes serán nuestro principal aliado a la hora de construir nuestro modelo. Las dividiremos según los recursos que nos hagan falta y las usaremos en toda la primera etapa del trabajo.
- **Placa Maix Bit.** Esta herramienta será el eje central de la segunda fase del trabajo. Incluye los diferentes módulos de procesamiento RISC-V junto al entorno de MicroPython que nos ayudará a cargar el modelo y ejecutarlo. La usaremos conectada a una *protoboard* la cual nos ayudará con la incorporación del resto de elementos electrónicos.



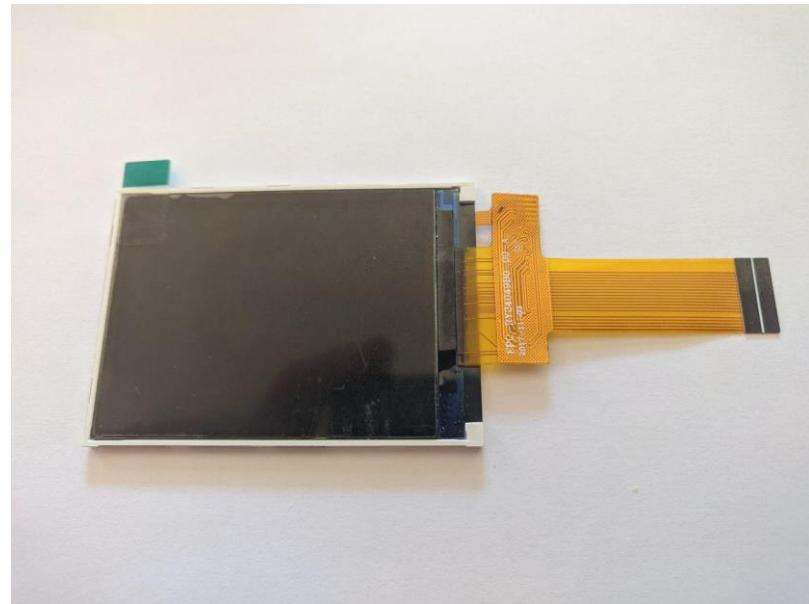
**Figura 53.** Placa Maix Bit

- **Sensor Óptico OV2640.** Para conseguir las imágenes que posteriormente clasificaremos según nuestro modelo necesitamos un tipo de sensor óptico. Nuestra placa es capaz de soportar cualquier tipo de sensor óptico de categoría OV2640.



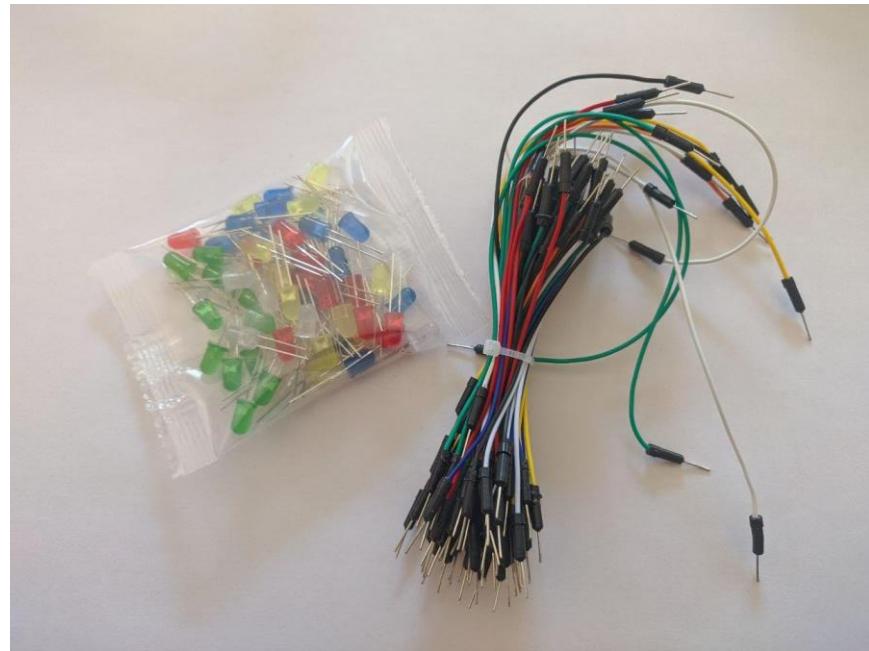
**Figura 54.** Sensor óptico OV2640 ojo de pez 2 Mpx

- **Pantalla LCD.** Con el fin de hacer un prototipo más visual sobre el que ejecutaremos el modelo, vamos a utilizar un pequeño LCD para ver las imágenes capturadas por el sensor junto a la clasificación obtenida.



*Figura 55. Pantalla LCD*

- **Cables y Leds.** Con la misma finalidad que la pantalla LCD incluiremos diferentes leds que nos indiquen de forma automática a que clase corresponde la imagen sin necesidad de visualizar la pantalla.



*Figura 56. Cables y Leds*

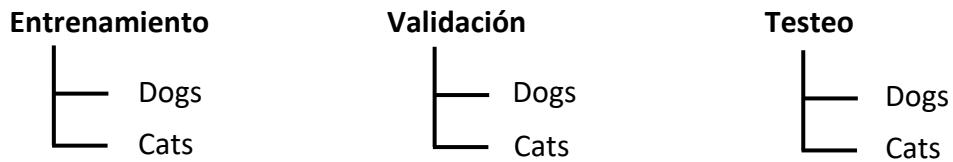
## 4.2 Pasos previos a la implementación

---

Como hemos mencionado anteriormente, es de vital importancia tener unos buenos datos de entrada que sean capaces de generar un buen modelo y que nos permitan hacer un estudio posterior a la generación del modelo.

El modelo que vamos a construir se trata de un clasificador entre imágenes de perros y gatos, para los cuales obtuvimos los datos de Kaggle según explicamos en la sección [3.4.2](#), indicando las diferentes secciones en las que debían separarse las imágenes.

En el *dataset* original obtenido en [Kaggle](#) las imágenes vienen divididas en carpetas de testeo y entrenamiento, en las cuales solo vienen las imágenes separadas por clases en las de entrenamiento. Estas imágenes vienen diferenciadas por el nombre del archivo, teniendo que clasificarlas en subcarpetas para usarlas en nuestro modelo. La estructura de carpetas es la siguiente:



Para conseguir esta distribución dividimos aleatoriamente las imágenes dadas como entrenamiento entre las dos primeras carpetas, consiguiendo todas las imágenes necesarias para nuestro proceso de entrenamiento. El número de imágenes usadas como validación es el correspondiente con el 25% del total de imágenes dadas para entrenamiento.

En las imágenes de testeo, al venir ambas clases mezcladas en la misma carpeta, tuvimos que manualmente mover una a una en las carpetas correspondiente hasta completar aproximadamente 6500 imágenes, eliminando algunas imágenes de ruido y otras inclasificables para hacer que los resultados sean los más precisos posibles.

Con este sistema de carpetas creado es mucho más sencillo usar los iteradores para entrenar el modelo. Estos iteradores automáticamente cogen como clases las diferentes subcarpetas dentro de la ruta que le indicamos, separando entre perros y gatos en nuestro caso.

Este proceso de clasificación en carpetas viene incluido en un notebook dentro del [repositorio del TFG](#).

## 4.3 Metodología

---

Con las imágenes clasificadas correctamente y una vez preparados todos los materiales podemos comenzar con la implementación como tal de las diferentes partes del proyecto.

Todos los pasos relacionados con la generación del modelo los realizaremos sobre [Jupyter Notebook](#) usando la versión 2.1.0 de Tensorflow. Estas herramientas las ejecutaremos a su vez desde [Anaconda](#), la cual incluye multitud de recursos para el desarrollo en Python.

Durante los diferentes pasos dados iremos justificando el uso de nuestras redes, así como algunas comparaciones con modelos e implementaciones que fuimos mejorando con el uso de nuevas técnicas de aprendizaje profundo.

#### 4.3.1 Generación del modelo

---

Nuestro principal objetivo es obtener un buen modelo el cual podamos, de manera sencilla, ejecutarlo en la placa. Dada la naturaleza del problema, la generación de un modelo que sea capaz de clasificar perros y gatos es tarea sencilla, sin embargo, generar lo cumpliendo los requisitos técnicos de la placa, se convierte en algo más complicado.

Como este problema es uno de los más conocidos en la clasificación usando *CNNs*, vamos a basarnos en los diferentes estudios realizados por diferentes usuarios sobre esta clasificación binaria para generar nuestro modelo.

La principal base la obtuvimos de la propia página de [TensorFlow](#), donde se hace un pequeño repaso de las herramientas disponibles en la herramienta, y de la [página tutorial](#) sobre este tipo de problema, haciendo un análisis de las diferentes formas que disponemos para afrontar el problema.

En ambas páginas usan una estructura simple combinando las capas de convolución y *pooling* varias veces antes de aplicar la red neuronal básica, dando unos resultados decentes en poco tiempo.

Como primera implementación nos basamos en los modelos vistos, generando un modelo el cual usa las técnicas para evitar el sobre aprendizaje como el *Dropout* o el *Data Augmentation*, consiguiendo un modelo sencillo y a su vez eficaz con este tipo de clasificación binaria.

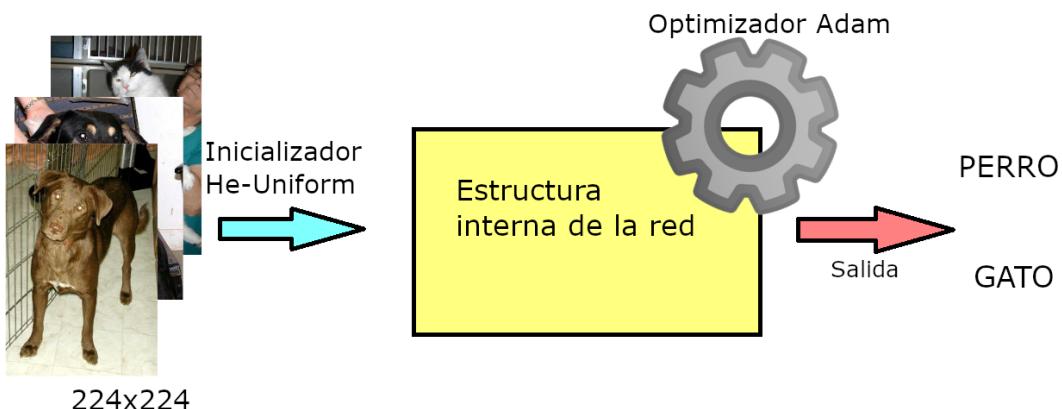
Model: "sequential"		
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 200, 200, 32)	896
max_pooling2d (MaxPooling2D)	(None, 100, 100, 32)	0
dropout (Dropout)	(None, 100, 100, 32)	0
conv2d_1 (Conv2D)	(None, 100, 100, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 50, 50, 64)	0
dropout_1 (Dropout)	(None, 50, 50, 64)	0
conv2d_2 (Conv2D)	(None, 50, 50, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 25, 25, 128)	0
dropout_2 (Dropout)	(None, 25, 25, 128)	0
flatten (Flatten)	(None, 80000)	0
dense (Dense)	(None, 128)	10240128
dropout_3 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 1)	129
Total params: 10,333,505		
Trainable params: 10,333,505		
Non-trainable params: 0		

**Figura 57.** Primera estructura generada

En este modelo y todos los que veremos posteriormente vamos a usar un tamaño de imagen de 224x224, el cual es el usado en redes populares como [ImageNet](#), la cual ya se ha conseguido implementar en nuestra placa como podemos ver en el [blog oficial](#) de la empresa.

Para inicializar las capas de los modelos que estudiaremos y usaremos la inicialización [He-Uniform](#), la cual es una de las mejores que podemos usar para este tipo de redes como vimos en el capítulo [3.4.3](#). Este tipo de inicialización es a su vez el usado en las páginas anteriormente vistas donde obtuvimos nuestro modelo base.

El último valor común a todas las redes que veremos en este capítulo va a ser la forma de optimizarlas, donde vamos a usar el optimizador incluido en Keras [Adam](#), que al igual que nuestro inicializador, vimos que era la mejor opción en el capítulo [3.4.3](#).

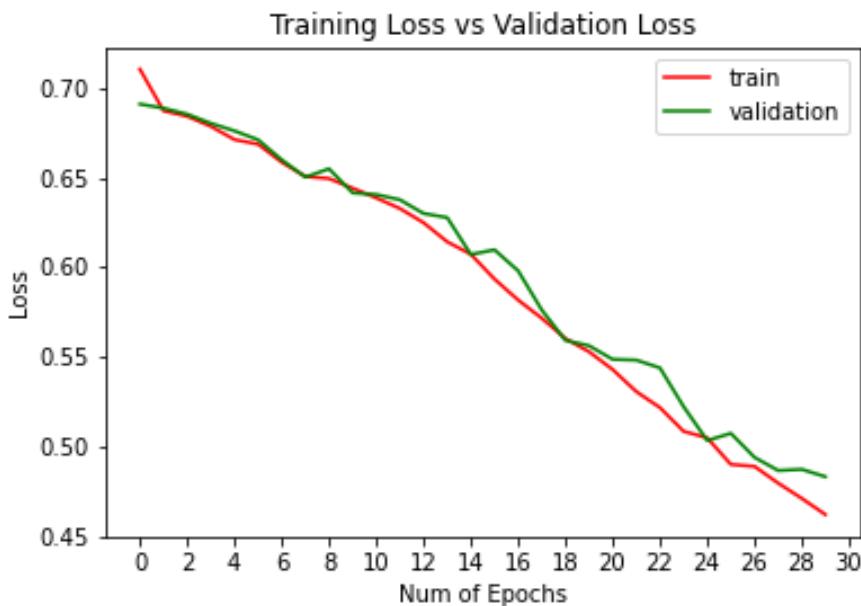


**Figura 58.** Estructura común a nuestros modelos

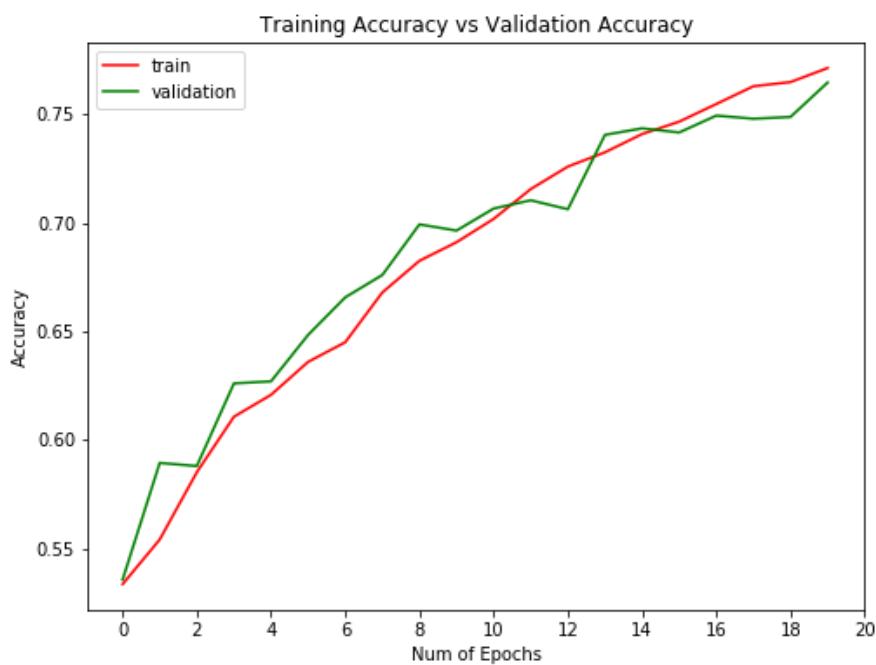
Con esta estructura conseguimos una base muy buena para resolver nuestro problema, siendo más fácil probar los diferentes tipos de capas y probando diferentes formas de estructurar la red para que sea compatible con la capacidad de la placa.

El primer modelo construido incluye únicamente tres capas de convolución y *max\_pooling* seguidas de un aplanamiento junto a una red clásica, la cual cuenta con más de 10 millones de parámetros, siendo completamente inviable para la incorporación en la placa.

Igualmente, para comprobar si el modelo conseguía unas gráficas de entrenamiento buenas probamos a ejecutar varias épocas sobre el modelo, generando los siguientes resultados:



**Figura 59.** Gráfica de perdida sobre el modelo base



**Figura 60.** Gráfica de precisión sobre el modelo base

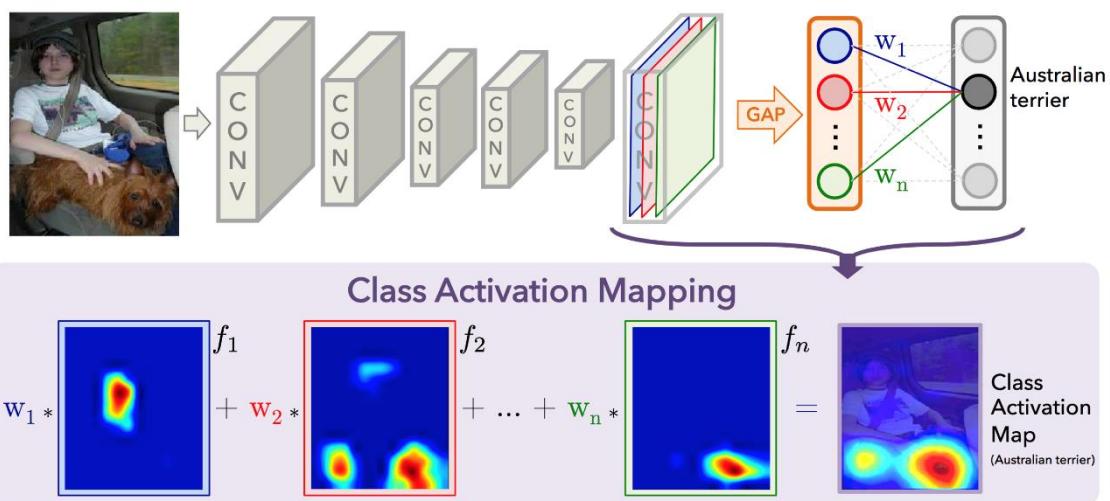
Ambas gráficas nos indican que, aunque el entrenamiento no ha llegado a su fin, los parámetros usados son correctos y una buena base para generar nuestro propio modelo, el cual debe, como mínimo, conseguir unos resultados similares a este primero.

El primer cambio evidente que debemos implementar es una reducción de parámetros, debiendo por tanto reducir la cantidad de parámetros de la capa densa de nuestra red, la cual incluye la mayoría de los parámetros del modelo.

Una de las formas de conseguir esto es reduciendo el tamaño de las imágenes de entrada en esta capa usando varias capas de convolución y *pooling* más, consiguiendo que las imágenes sean de un tamaño inferior y que por lo tanto no necesiten tantos parámetros para ser analizadas.

Para ahorrar a su vez un poco más de parámetros y dado que no nos interesan las diferentes características obtenidas en cada una de las capas, sino que nos interesa únicamente la clasificación final, vamos a usar una nueva capa llamada *GlobalAveragePooling2D (GAP)*.

Esta capa coge la media de todos los mapas de características presentes en cada capa de convolución y aplicándole posteriormente una función de activación *Softmax*, el cual convierte los valores en probabilidades de 0 a 1. Con estas probabilidades escoge la que mejor recoge las características de la imagen, usándola para clasificar entre clases.



**Figura 61.** Funcionamiento de la capa de GAP

Con el uso de estas técnicas elaboramos un nuevo modelo, el cual reduce al máximo los parámetros usados y consigue unos resultados muy satisfactorios, siendo la opción más viable para incorporar en nuestra placa.

Como complemento del *dropout* usado anteriormente hemos añadido las técnicas de *data augmentation*, consiguiendo aumentar el número de ejemplos usados para el entrenamiento y por tanto el número de posibles situaciones a analizar.



**Figura 62.** Data Augmentation aplicado en nuestro modelo.

En nuestro caso, la forma de modificar las imágenes originales va a ser volteando la imagen hacia alguno de los lados y desplazándolas tanto verticalmente como horizontalmente en alguna de las cuatro coordenadas, consiguiendo los resultados vistos en la figura [62](#).

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 224, 224, 16)	448
average_pooling2d (AveragePo	(None, 112, 112, 16)	0
conv2d_1 (Conv2D)	(None, 112, 112, 32)	4640
average_pooling2d_1 (Average	(None, 56, 56, 32)	0
conv2d_2 (Conv2D)	(None, 56, 56, 64)	18496
average_pooling2d_2 (Average	(None, 28, 28, 64)	0
conv2d_3 (Conv2D)	(None, 28, 28, 128)	73856
average_pooling2d_3 (Average	(None, 14, 14, 128)	0
conv2d_4 (Conv2D)	(None, 14, 14, 256)	295168
average_pooling2d_4 (Average	(None, 7, 7, 256)	0
global_average_pooling2d (G	(None, 256)	0
flatten (Flatten)	(None, 256)	0
dropout (Dropout)	(None, 256)	0
dense (Dense)	(None, 32)	8224
dense_1 (Dense)	(None, 1)	33
<hr/>		
Total params:	400,865	
Trainable params:	400,865	
Non-trainable params:	0	

**Figura 63.** Estructura de red reducida

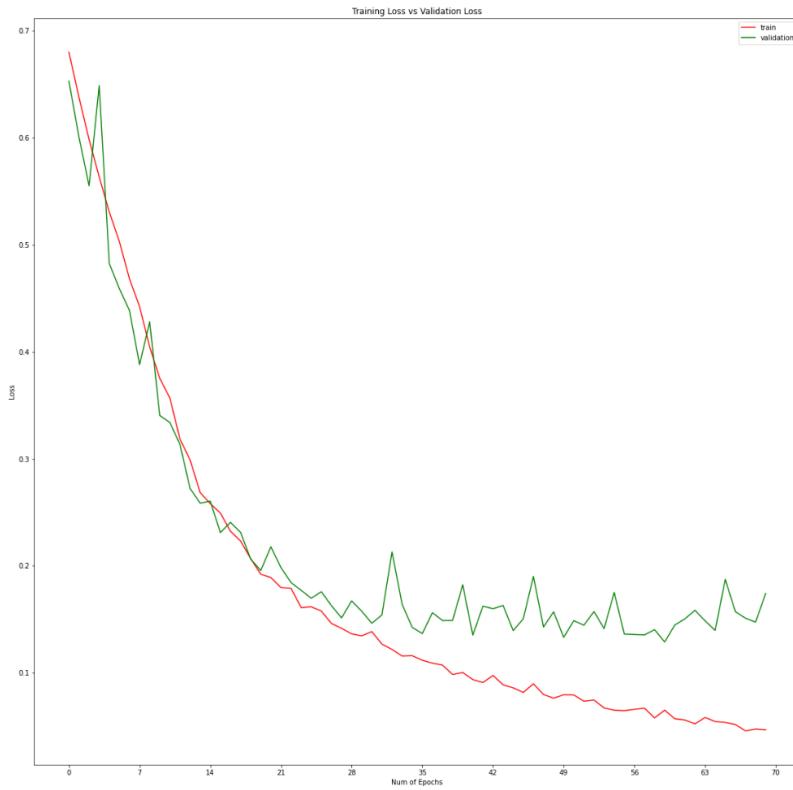
Al igual que la primera estructura generada, esta acaba con una única neurona que será la que indica a cuál de las dos clases pertenece. Esta tiene una función de activación **sigmoide**, tomando como gato a los valores menores de 0.5 y como perro al resto.

Este tipo de clasificación binaria solo se puede usar en problemas que cuentan con dos clases, teniendo únicamente que clasificar entre dos opciones contrapuestas, las cuales son fácilmente clasificables usando la función sigmoide.

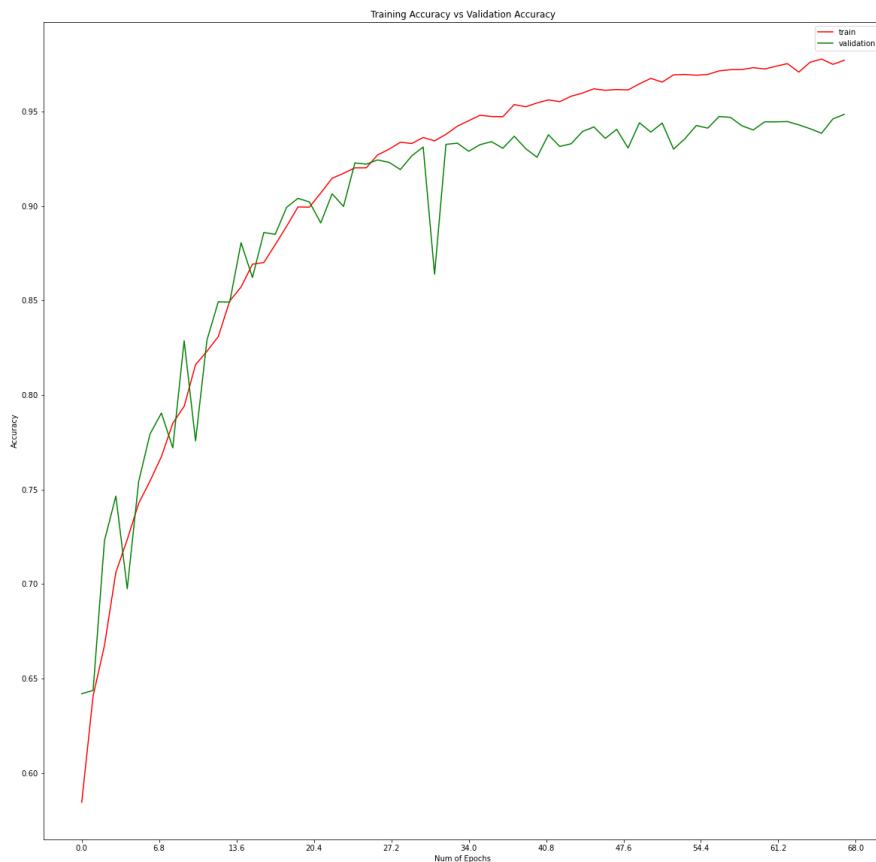
Para conseguir los mejores resultados posibles del modelo vamos a hacer uso de la herramienta [Callback](#) proporcionada por *Tensorflow*, la cual nos ayudará a controlar el proceso de aprendizaje y a guardar en cada momento el mejor modelo conocido hasta el momento.

Nuestro *Callback* está configurado para parar si no mejora la perdida en diez épocas seguidas, parando el entrenamiento en ese punto y devolviendo el mejor modelo encontrado hasta ahora según este valor.

Los resultados obtenidos durante el entrenamiento han sido los siguientes:



**Figura 64.** Gráfica de pérdida del modelo reducido (Aprox. 0,14 min)



**Figura 65.** Gráfica de precisión del modelo reducido (Aprox. 0,95 max)

A diferencia de las primeras gráficas que analizamos, estas demuestran que el modelo ha convergido en unos valores muy buenos para clasificar las imágenes de nuestro problema.

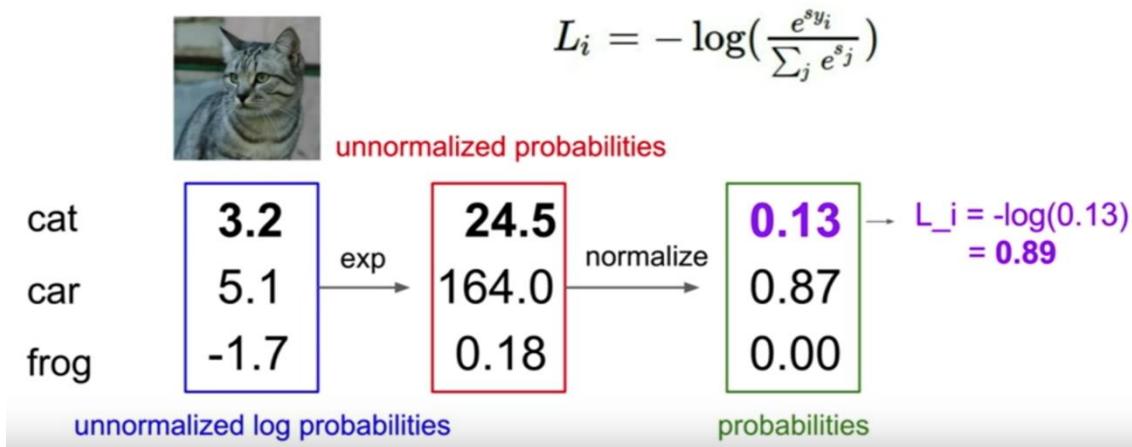
Con una precisión de aproximadamente un 95% y un valor de pérdida de 14% demostramos que los elementos utilizados para generar el modelo son los óptimos para este problema, pudiendo en consecuencia, continuar con la optimización de este modelo para su posterior incorporación a nuestra placa.

En nuestra placa no podemos cargar problemas de naturaleza binaria para su clasificación, sino que en su lugar debemos utilizar el “*class mode*” como “*categorical*”, es decir, el modelo tendrá como salida tantas neuronas como número de clases de las que disponga, calculando en cada una de ellas el porcentaje de parecido con cada clase en cuestión.

Este modo se debe indicar a la hora de crear los iteradores que van a recorrer las imágenes de entrada para conseguir entrenar el modelo. Con esta opción activada conseguiremos indicar a Keras que nuestras clases van a estar dadas con probabilidades, necesitando para ello una neurona para cada clase.

Para conseguir que estas neuronas tengan este valor aplicaremos la función de activación *softmax* a la salida, la cual, como ya hemos comentado anteriormente, reparte las probabilidades que tienen una imagen de pertenecer a cada una de las clases (en nuestro caso)

## Softmax Classifier (Multinomial Logistic Regression)



**Figura 66.** Funcionamiento del clasificador SoftMax

Este pequeño cambio solo supone cambiar la forma en la que tratamos los iteradores que van a controlar las imágenes, dividiéndolo ahora en dos clases independientes y no controlando el valor de una única neurona.

Usando este tipo de entrenamiento y este tipo de distribución en las imágenes podemos tener controlado de una mejor manera el porcentaje de pertenencia a cada una de las clases, pudiendo hacer un posterior estudio mucho más preciso del clasificador.

En la estructura del modelo debemos añadir una neurona más a la capa de salida para tener una por clase y así poder aplicar la función de activación anteriormente mencionada, consiguiendo así el modelo compatible con nuestra placa.

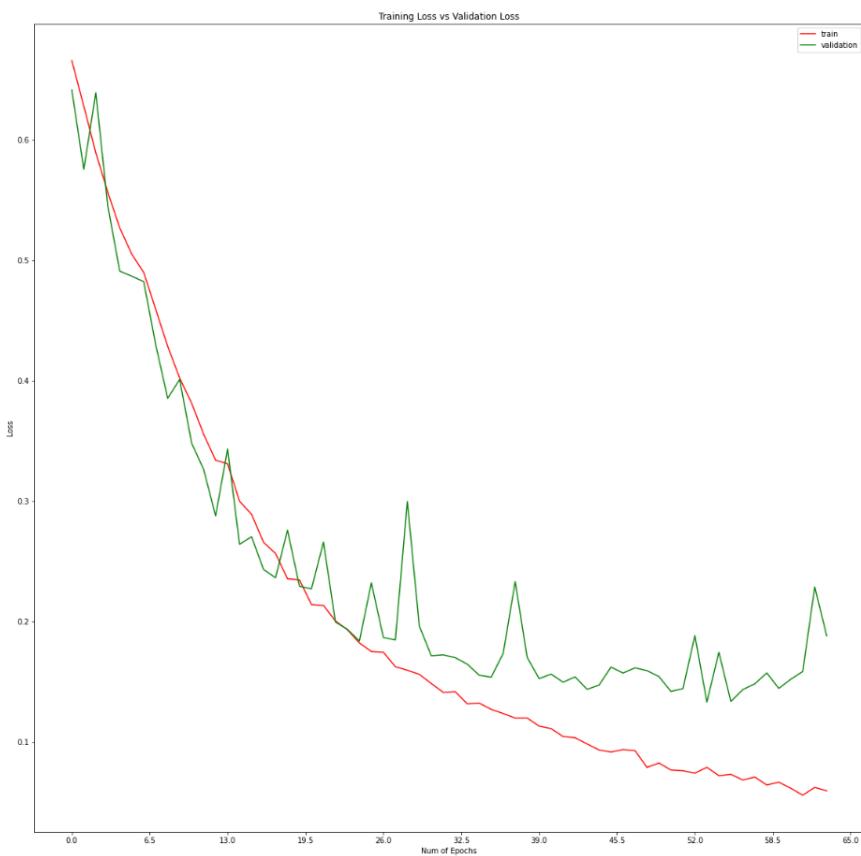
Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 224, 224, 16)	448
average_pooling2d (AveragePo	(None, 112, 112, 16)	0
conv2d_1 (Conv2D)	(None, 112, 112, 32)	4640
average_pooling2d_1 (Average	(None, 56, 56, 32)	0
conv2d_2 (Conv2D)	(None, 56, 56, 64)	18496
average_pooling2d_2 (Average	(None, 28, 28, 64)	0
conv2d_3 (Conv2D)	(None, 28, 28, 128)	73856
average_pooling2d_3 (Average	(None, 14, 14, 128)	0
conv2d_4 (Conv2D)	(None, 14, 14, 256)	295168
average_pooling2d_4 (Average	(None, 7, 7, 256)	0
global_average_pooling2d (Gl	(None, 256)	0
flatten (Flatten)	(None, 256)	0
dropout (Dropout)	(None, 256)	0
dense (Dense)	(None, 32)	8224
dense_1 (Dense)	(None, 2)	66
<hr/>		
Total params:	400,898	
Trainable params:	400,898	
Non-trainable params:	0	

**Figura 67.** Estructura del modelo final

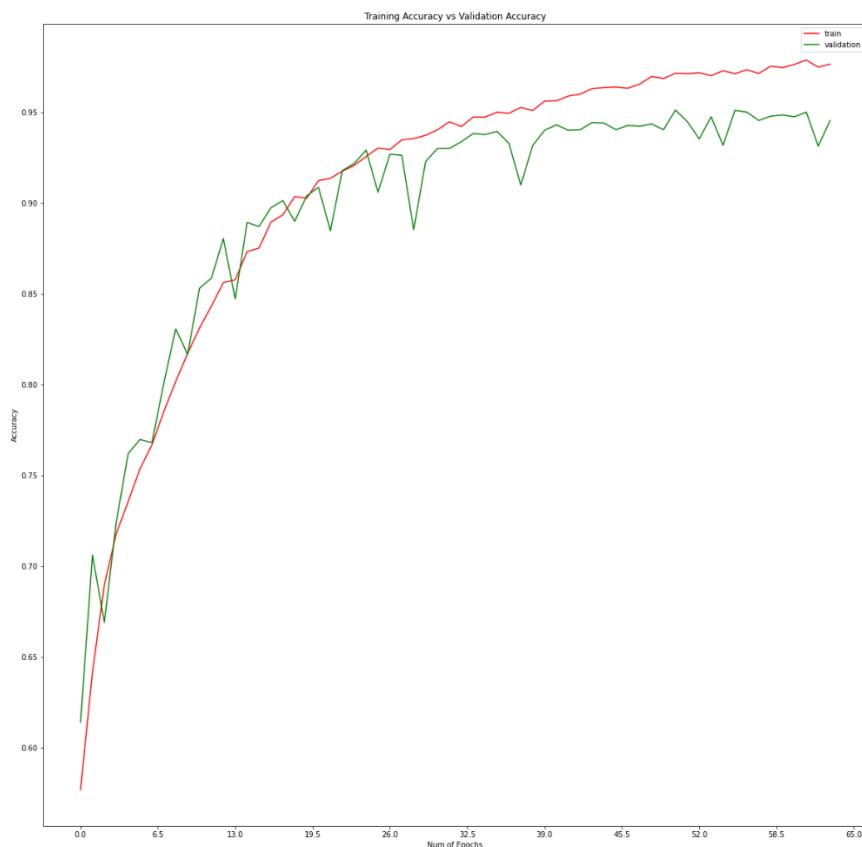
Este modelo tiene un número de parámetros perfecto para la incorporación en la placa y a su vez cuenta con la función de activación necesaria para hacerlo funcionar de forma correcta.

Las gráficas obtenidas en este modelo siguen el mismo comportamiento de las anteriores, consiguiendo ver la separación entre las líneas de validación y entrenamiento, dando por concluido el entrenamiento.

Para continuar guardando siempre el mejor modelo obtenido en todo el entrenamiento vamos a usar un *callback* de 10 con el valor de perdida al igual que el modelo anterior.



**Figura 68.** Gráfica de pérdida en el modelo final (Aprox. 0,15 min)



**Figura 69.** Gráfica de precisión en el modelo final (Aprox. 0,95 max)

Estas gráficas nos indican la veracidad del modelo, dando a ver que en la mayoría de las imágenes que se le presenten va a dar un resultado correcto, dando mucha fiabilidad y confianza a este.

Este modelo cumple todos los requisitos requeridos por la placa para conseguir una ejecución correcta, por lo tanto, es un buen candidato para ser usado en la placa, quedando pendiente el estudio de los resultados con las imágenes de testeo, el cual veremos en el capítulo a continuación.

El código donde procesamos el tratamiento de las imágenes con iteradores y el entrenamiento de la red final usada lo hemos incorporado en nuestro repositorio de GitHub con el nombre de “CNNs 95% precisión”.

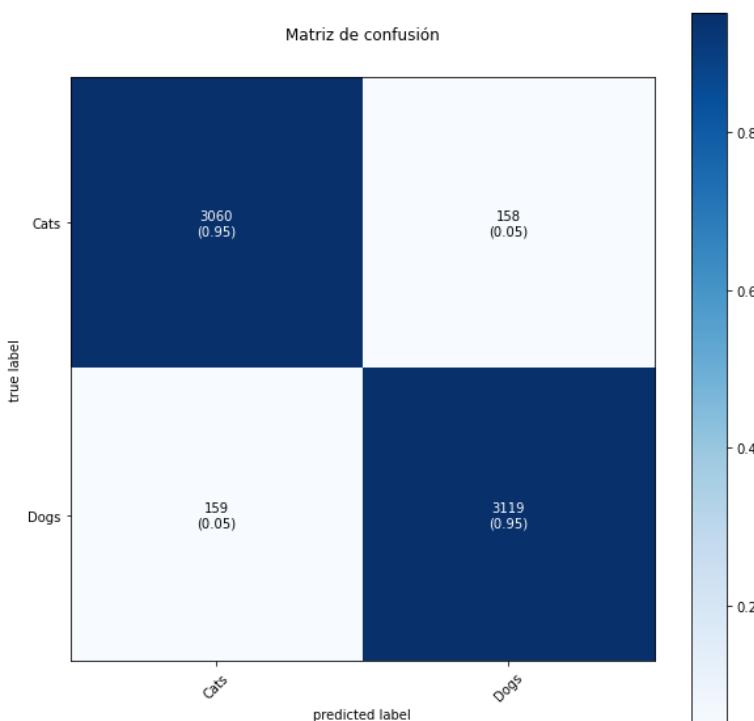
#### 4.3.2 Estudio de resultados

---

Una vez obtenido un modelo valido, debemos comprobar cómo funciona con imágenes nunca vistas por la red, de forma que podamos corroborar los datos obtenidos durante el entrenamiento con los obtenidos al aplicar la clasificación sobre este tipo de imágenes.

En esta fase es donde entran en juego las imágenes de testeo, las cuales no han sido usadas en ningún proceso del entrenamiento y que están listas para ser clasificadas por el modelo construido durante la etapa anterior.

Para realizar el estudio vamos a aplicar las diferentes métricas que describimos en el apartado [3.4.4](#) de la memoria, los cuales comienzan por hacer una matriz de confusión donde nos muestren los valores de FP, FN, TP y TN. Esta matriz se va a construir usando las imágenes de testeo clasificadas por carpetas, las cuales se encuentran dentro del *dataset* en la carpeta “*test\_classes*”.

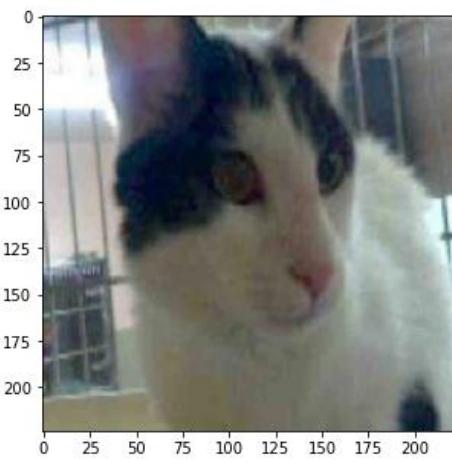


**Figura 70.** Matriz de confusión generada

En la matriz podemos ver un valor numérico elevado en la diagonal, indicando que las clases reales con las predichas son las correctas, teniendo un valor muy bajo en la diagonal contraria, que indica los valores de FP y FN.

Con estos valores de precisión para cada una de las clases (aprox. 0.95) podemos afirmar la robustez del modelo sobre las diferentes imágenes existentes en el *dataset* de testeo, comportándose de manera similar que con las imágenes de validación vistas durante el entrenamiento.

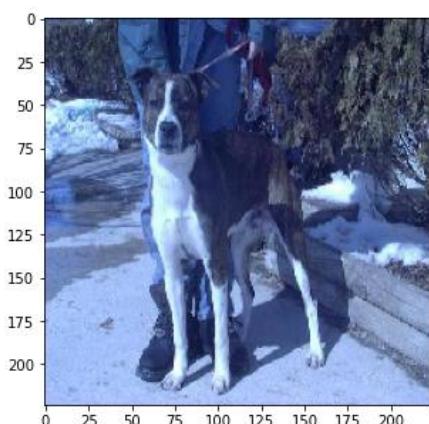
Durante las pruebas con las imágenes de testeo nos damos cuenta de que en la mayoría de las fotos el porcentaje de correspondencia es muy cercano al 100% en alguno de los extremos, clasificando de forma casi segura las imágenes en la clase correcta. Algunos ejemplos son:



Probabilidad Gato: 0.71543473  
Probabilidad Perro: 0.2845653

Probabilidad máxima: 0.71543473  
Clasificado como: Gato

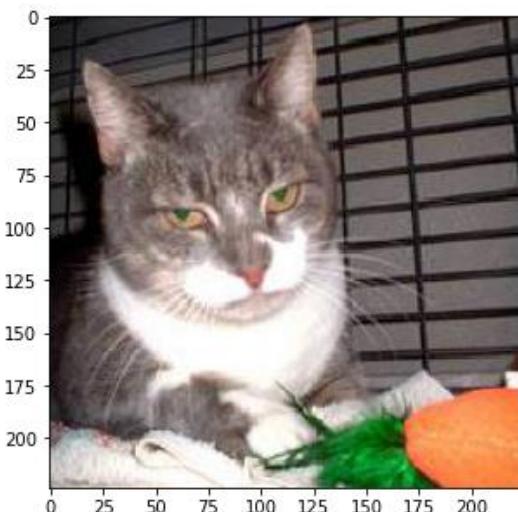
**Figura 71.** Imagen de gato predicha correctamente con porcentaje del 71%



Probabilidad Gato: 0.012631667  
Probabilidad Perro: 0.9873683

Probabilidad máxima: 0.9873683  
Clasificado como: Perro

**Figura 72.** Imagen de perro predicha correctamente con porcentaje del 98%



Probabilidad Gato: 0.99999714  
Probabilidad Perro: 2.8501147e-06

Probabilidad máxima: 0.99999714  
Clasificado como: Gato

**Figura 73.** Imagen de gato predicha correctamente con porcentaje del 99,9%

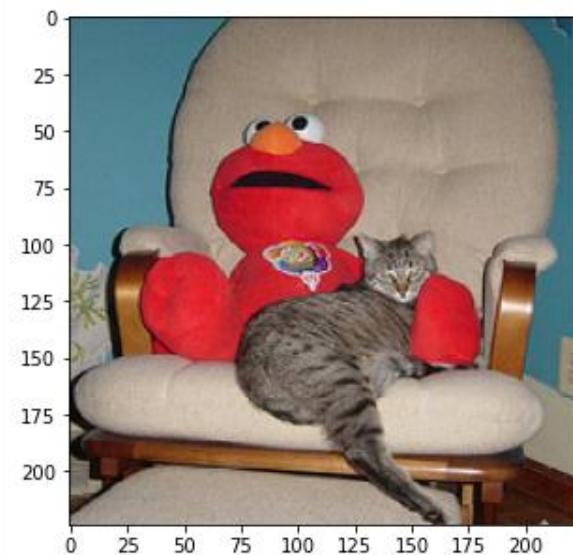
Como hemos podido ver en la matriz de confusión, la inmensa mayoría de las imágenes vienen clasificadas de forma correcta, sin embargo, existen algunas que se han clasificado incorrectamente, podemos ver algunos de esos ejemplos:



Probabilidad Gato: 0.7882892  
Probabilidad Perro: 0.21171075

Probabilidad máxima: 0.7882892  
Clasificado como: Gato

**Figura 74.** Imagen de perro clasificada incorrectamente con un porcentaje del 78%



Probabilidad Gato: 0.24040695

Probabilidad Perro: 0.759593

Probabilidad máxima: 0.759593

Clasificado como: Perro

**Figura 75.** Imagen de gato clasificada incorrectamente con porcentaje del 75,9%

En las imágenes que dan incorrectamente normalmente el porcentaje de pertenencia no supera el 80%, indicando que la red en algunos momentos se encuentra indecisa frente algunas imágenes que no pertenecen correctamente a sus clases.

Vistos algunos ejemplos y la matriz de confusión podemos calcular los datos de control para conocer la fiabilidad de la red teniendo en cuenta que nuestra clase negativa son los gatos y nuestra clase positiva son los perros.

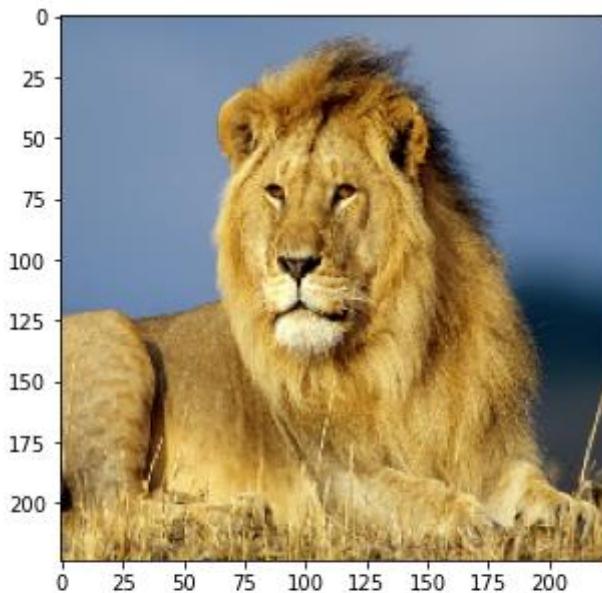
	Gatos (Negativa)	Perros (Positiva)
<b>Sensitivity</b>	0.95090118	0.95149481
<b>Specificity</b>	0.95149481	0.95090118
<b>Positive Predictive Value (PPV)</b>	0.95060578	0.95178517
<b>Negative Predictive Value (NPV)</b>	0.95178517	0.95060578
<b>False Positive Rate (FPR)</b>	0.04850519	0.04909882
<b>False Negative Rate (FNR)</b>	0.04909882	0.04850519
<b>False Discovery Rate (FDR)</b>	0.04939422	0.04821483
<b>ACCURACY</b>	<b>0.95120074</b>	

**Figura 76.** Tabla con datos de comportamiento de la red

Con los datos obtenidos del modelo usando las imágenes de testeo podemos concluir que nuestro modelo es un buen clasificador de imágenes entre perros y gatos, teniendo un buen porcentaje de imágenes clasificadas correctamente y un bajo porcentaje en los datos que indican una clasificación errónea.

Según la tabla, es difícil equivocar un perro con un gato o viceversa, sin embargo, todo esto trata con imágenes dentro de un entorno cerrado donde solo nos encontramos con imágenes o de una clase o de otra, sin contar otras posibilidades.

Si por ejemplo pasáramos imágenes que no son de ninguna de las dos clases a la red, esta clasificaría de manera totalmente aleatoria sobre alguna de ambas clases, podemos ver algunos ejemplos:



Probabilidad Gato: 0.4406031

Probabilidad Perro: 0.5593969

Probabilidad máxima: 0.5593969

Clasificado como: Perro

**Figura 77.** Imagen de león pasada por el modelo

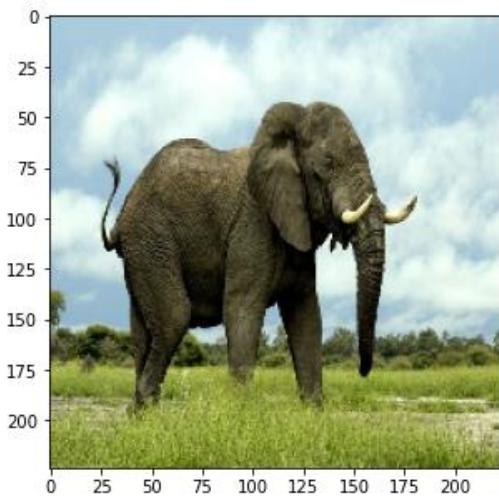
En la figura 77 podemos ver una de las imágenes que más información nos puede dar sobre la red. Esta imagen se encuentra muy cercana a la frontera de decisión de la red, la cual no es capaz de clasificar la imagen de entrada sobre ninguna de las clases de las cuales ha aprendido.

Este fenómeno es correcto ya que en ningún momento le hemos enseñado a la red lo que es un “león” y buscando características parecidas a los gatos y a los perros se encuentra frente a la incertidumbre, dando lugar a los datos de salida de la red.

Es muy importante conocer los límites de la red junto al uso de esta. Conociendo estos límites podemos sacar mucho más partido a nuestro modelo, pudiendo realizar un estudio más realista sobre los resultados obtenidos y, sobre todo, sabiendo a ciencia cierta los resultados aproximados que debe dar y a qué aplicaciones se podría aplicar.

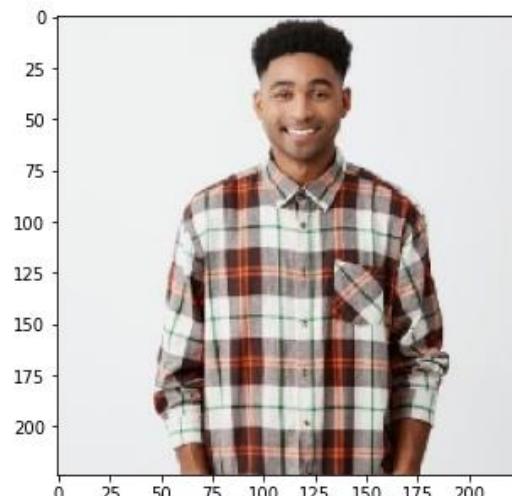
Nuestro modelo está diseñado para clasificar entre estas dos clases, siendo inservible para cualquier otro tipo de imágenes, pudiendo dar un resultado aleatorio hacia cualquiera de las dos.

En el ejemplo del león podríamos indicar que es una imagen desconocida debido a los porcentajes de pertenencia tan bajos, sin embargo, en otro tipo de imágenes estos porcentajes son muy elevados hacia alguna de las dos clases, siendo imposible tratarlas como imágenes desconocidas.



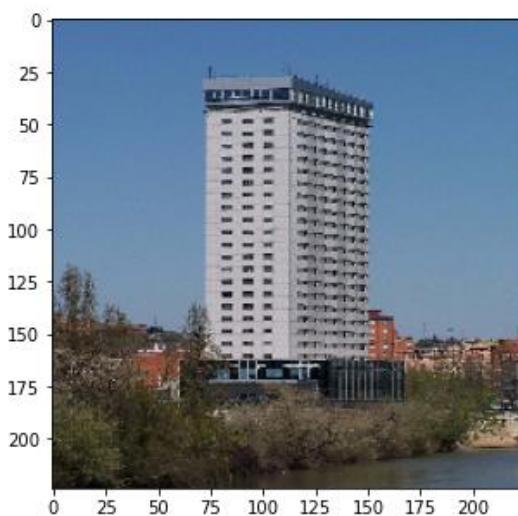
Probabilidad Gato: 2.4528083e-05  
Probabilidad Perro: 0.99997544

Probabilidad máxima: 0.99997544  
Clasificado como: Perro



Probabilidad Gato: 0.06433058  
Probabilidad Perro: 0.9356695

Probabilidad máxima: 0.9356695  
Clasificado como: Perro



Probabilidad Gato: 0.21802986  
Probabilidad Perro: 0.78197014

Probabilidad máxima: 0.78197014  
Clasificado como: Perro



Probabilidad Gato: 0.9722387  
Probabilidad Perro: 0.027761318

Probabilidad máxima: 0.9722387  
Clasificado como: Gato

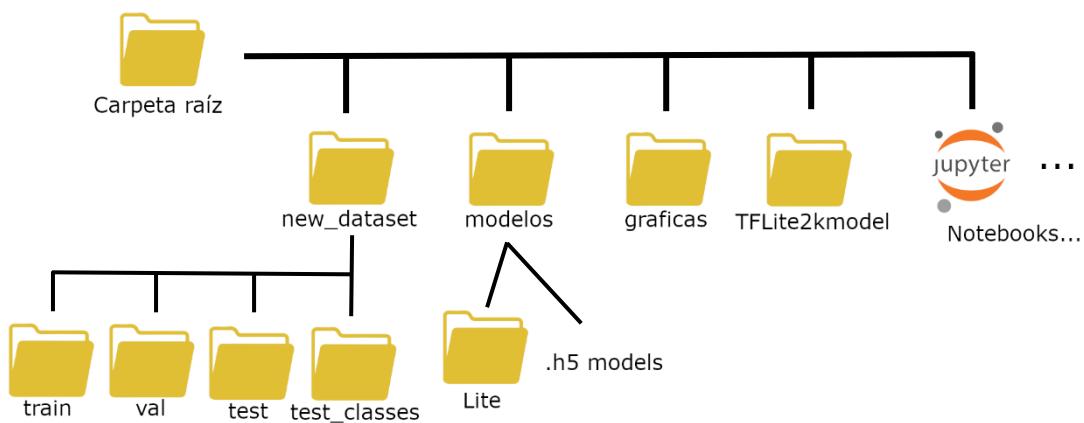
**Figura 78.** Imágenes de clases desconocidas

Para concluir podemos decir que nuestro modelo es un buen clasificador entre las clases para las que ha sido entrenado pero que, sin embargo, también es capaz de clasificar imágenes de clases diferentes hacia alguna de las dos clases sin indicar desconocimiento de estas, debiendo tener cuidado a la hora de usarlo en entornos no controlados.

#### 4.3.3 Exportación del modelo a la placa

---

Antes de comenzar, como vamos a hacer uso de varias carpetas las cuales son importante mantener de una forma u otra parecidas, vamos a hacer un repaso de como tenemos repartido nuestro entorno de trabajo, el cual será el que haga que funcionen todos los notebooks expuestos en nuestro repositorio de GitHub. El esquema de nuestro entorno de trabajo es el siguiente:



**Figura 79.** Estructura de nuestro entorno de trabajo

Este entorno es importante mantenerlo de manera parecida para que los enlaces expuestos en los notebooks no sufran mucha variación, siendo así más fácil su entendimiento y su aplicación.

Dentro de las carpetas de los *datasets* se encuentran dos subcarpetas con las imágenes de los perros y de los gatos (excepto en la de “test”), las cuales son la referencia que usa keras para realizar la división entre clases a la hora de entrenar el modelo.

Las demás carpetas las iremos explicando a lo largo de este capítulo junto a los pasos que debemos ir siguiendo para conseguir nuestro objetivo. Continuemos por tanto con los pasos necesarios para correr nuestro modelo en nuestro dispositivo.

Una vez tenemos este modelo completamente analizado debemos transformarlo de forma que podamos incluirlo en nuestra placa, la cual necesita un modelo cuya extensión sea “.kmodel”.

El primer paso que debemos dar es la transformación de nuestro modelo secuencial a un modelo de Tensorflow Lite. Esto se puede conseguir siguiendo los pasos proporcionados desde la [página oficial](#) de manera sencilla.

En nuestro repositorio incluimos un pequeño notebook el cual se encarga de coger un modelo **.h5** guardado en la carpeta **modelos** que ha sido generado por nuestra red y lo transforma en un modelo **.tflite** guardándola en la carpeta **modelos/Lite**.

Una vez tenemos nuestro modelo transformado a la versión lite necesitamos conseguir la versión compatible con la placa volviendo a transformar el resultado, pero esta vez hacia un modelo “.kmodel”.

Para poder realizar este paso es necesario tener el sistema operativo Ubuntu instalado o, en su defecto, la aplicación [Ubuntu de la Microsoft Store](#). Esta última incluye una ventana de comandos igual a la terminal del sistema operativo Ubuntu y nos permite ejecutar los lotes de instrucciones de manera similar a como lo haríamos directamente desde el original.



**Figura 80.** Ventana de comandos de Ubuntu Win10

Existe un [tutorial](#) de la propia página de Ubuntu explicando como instalar de forma correcta el sistema operativo para evitar errores de ejecución y fallos a la hora de ejecutar los ficheros por lotes.

La carpeta [TFLite2kmodel](#) incluye todos los ficheros necesarios para realizar la transformación de nuestro modelo de forma sencilla. Estos ficheros han sido obtenidos del [repositorio oficial](#) de la empresa fabricante de la placa y han sido modificados para su funcionamiento desde este Ubuntu ejecutado desde Windows 10.

La forma de realizar la transformación es copiando el modelo [.tflite](#) que queramos transformar a la raíz de la carpeta [TFLite2kmodel](#) y abrir nuestro terminal de Ubuntu. Dentro de este terminal llegamos a la raíz de esta carpeta y ejecutamos el archivo [tflite2kmodel.sh](#) junto al nombre del modelo [.tflite](#). Tras su ejecución se nos generará un archivo [.kmodel](#) en la carpeta raíz.

Como podemos ver en la figura 81 al ejecutar la instrucción aparecen las diferentes capas que componen nuestro modelo, comprobando que todo el proceso de transformación ha ido correctamente. Una de las comprobaciones previas al siguiente paso es ver que la capa de salida tenga la función de activación *Softmax* con tantas neuronas como número de clases a clasificar, ya que, si no, el modelo no se podrá ejecutar correctamente en el interior de la placa.

```

daniel@DESKTOP-CQAU1TM:/mnt/c/Users/Danie/Desktop/PRUEBAS_TFG/CNN_PERROS_GATOS/TFLite2Kmodel$ ls
BM_Categorical.tflite ckptpb.py get_nncase.sh pb2tflite.sh tflite2kmodel.sh
LICENSE gen_ckpt_graph.py images ncl pbtxt2pb.py tflite2pb.sh
README.md gen_pb_graph.py keras_to_tensorflow.py pb2pbtxt.py tflite2addpad.sh workspace
daniel@DESKTOP-CQAU1TM:/mnt/c/Users/Danie/Desktop/PRUEBAS_TFG/CNN_PERROS_GATOS/TFLite2Kmodel$ ./tflite2kmodel.sh BM_Categorical.tflite
usage: ./tflite2kmodel.sh xxx.tflite
2020-07-02 13:07:37.877227: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not compiled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
0: InputLayer -> 1x3x224x224
1: K210Conv2d 1x3x224x224 -> 1x16x112x112
2: K210Conv2d 1x16x112x112 -> 1x32x56x56
3: K210Conv2d 1x32x56x56 -> 1x64x28x28
4: K210Conv2d 1x64x28x28 -> 1x128x14x14
5: K210Conv2d 1x128x14x14 -> 1x256x7x7
6: Dequantize 1x256x7x7 -> 1x256x7x7
7: GlobalAveragePool 1x256x7x7 -> 1x256x1x1
8: Reshape 1x256x1x1 -> 1x256
9: Quantize 1x256 -> 1x256
10: K210AddPadding 1x256 -> 1x256x4x4
11: K210Conv2d 1x256x4x4 -> 1x32x4x4
12: K210Conv2d 1x32x4x4 -> 1x2x4x4
13: K210RemovePadding 1x2x4x4 -> 1x2
14: Dequantize 1x2 -> 1x2
15: Softmax 1x2 -> 1x2
16: OutputLayer 1x2
KPU memory usage: 2097152 B
Main memory usage: 62720 B

```

**Figura 81.** Transformación a kmodel desde el terminal de Ubuntu

Una vez hemos conseguido el modelo compatible con nuestra placa necesitamos cargar estos datos en el interior del dispositivo. Una vez desde el interior ya podemos elaborar los scripts de MicroPython necesarios para hacer funcionar el modelo y conseguir clasificar las imágenes capturadas por el sensor.

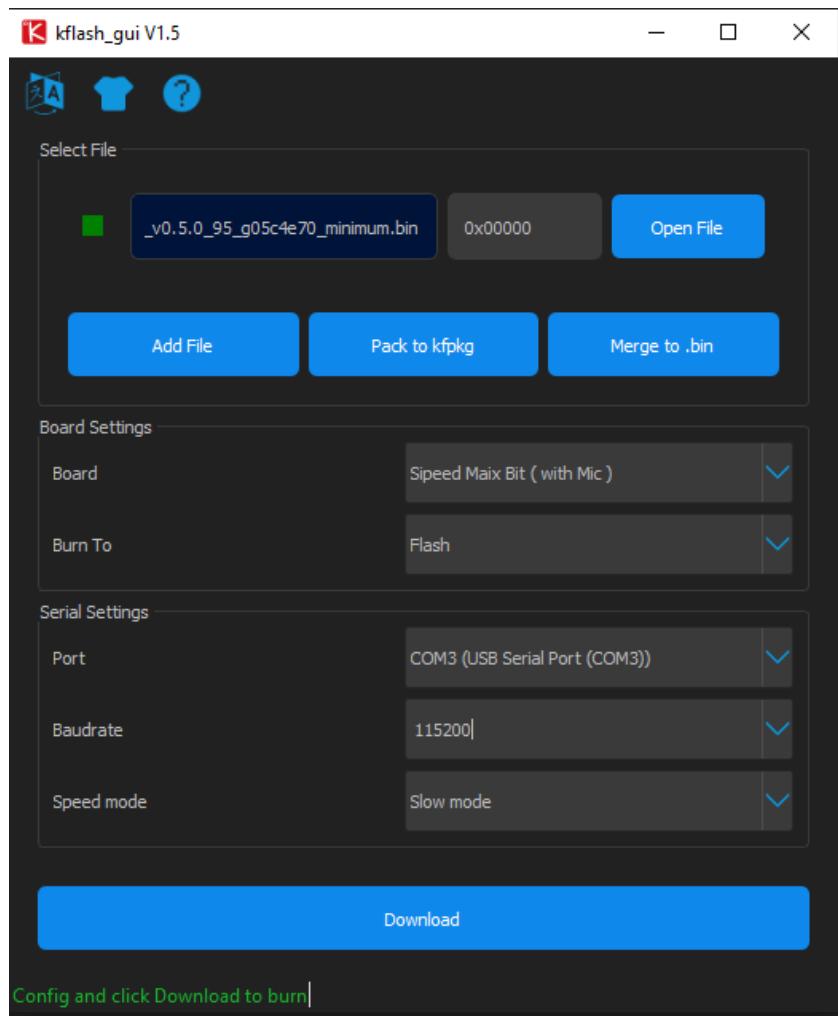
Antes de cargar el modelo es importante tener el firmware de la placa cargado en el interior. Este firmware incluye las diferentes librerías que usaremos durante la ejecución, así como el entorno de MicroPython junto alguna de sus funciones más usadas.

Como hemos comentado en capítulos anteriores, existen varias versiones de Firmware, las cuales pueden ser descargadas desde el [repositorio oficial](#) para un propósito diferente. En nuestro proyecto usaremos la versión 0.5 *mínimum*, la cual no incluye todas las funciones ni compatibilidad con el entorno gráfico [MaixPyIDE](#), pero si incluye más espacio para cargar un modelo de mayor capacidad, que es lo que más nos interesa.

Para poder cargar este firmware en la placa necesitamos el programa K-Flash GUI, el cual se puede descargar desde el [repositorio oficial](#) de la empresa. Este programa es muy sencillo de usar y lo único que debemos configurar es el puerto en el que está conectado nuestra placa.

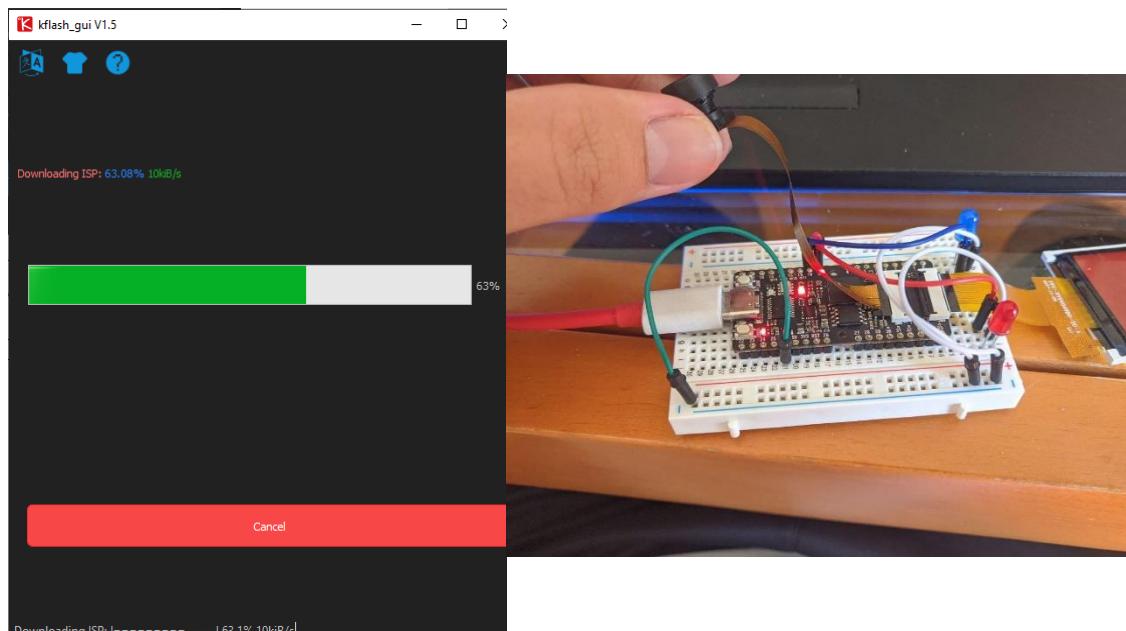
En nuestro caso, la placa que estamos usando es la denominada como “*Sipeed Maix Bit (with Mic)*” y está conectada al puerto COM3. Para que se cargue el firmware principal debemos indicar la dirección de memoria 0x000000, que será la cargada inicialmente al enchufar la placa a la corriente.

Al programa, además de todo lo anterior, tenemos que indicarle el número de señales por segundo (*Baudrate*) que vamos a pasarle a nuestro dispositivo. Nuestro modelo únicamente consigue cargar los archivos usando el número mínimo de *Baudrate*, siendo un valor de 115200.



**Figura 82.** Interfaz K-Flash GUI configurada

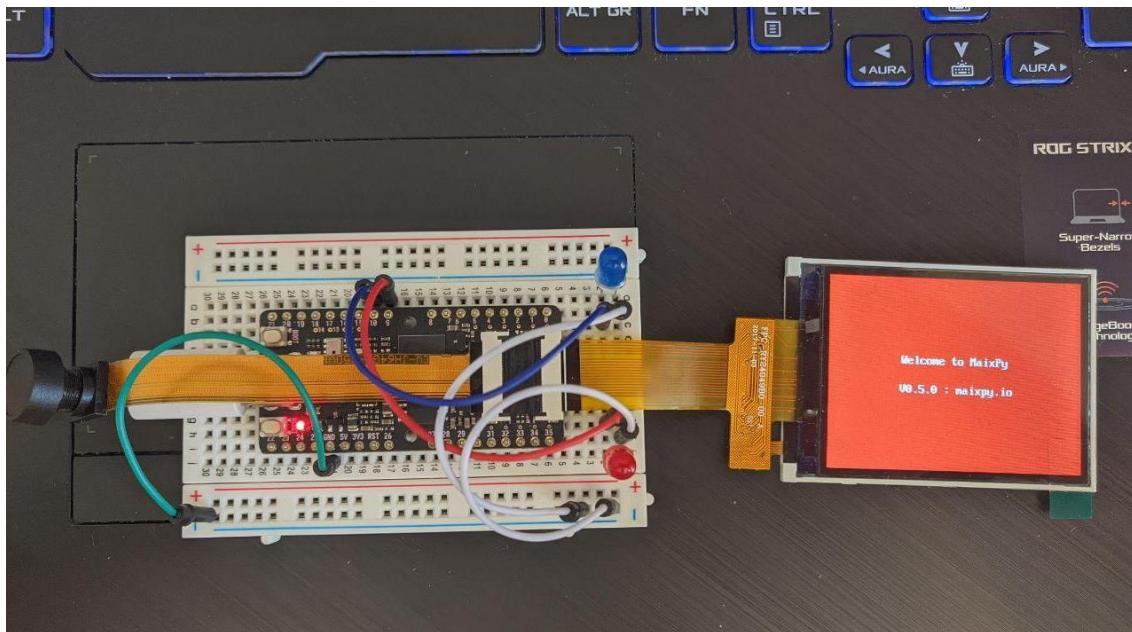
Tras configurar todos los parámetros solo debemos darle a “Download” y esperar a que el programa finalice la carga.



**Figura 83.** Proceso de carga en la placa

Como podemos observar en las figuras [83](#), para comprobar que el proceso se está realizando correctamente debemos ver encendidos ambos leds en la placa junto a la barra de carga en la interfaz del programa.

Al terminar de cargar nuestro firmware la placa se reiniciará automáticamente, coloreando la pantalla de rojo e indicando la versión del firmware (a partir de la versión 0.5).



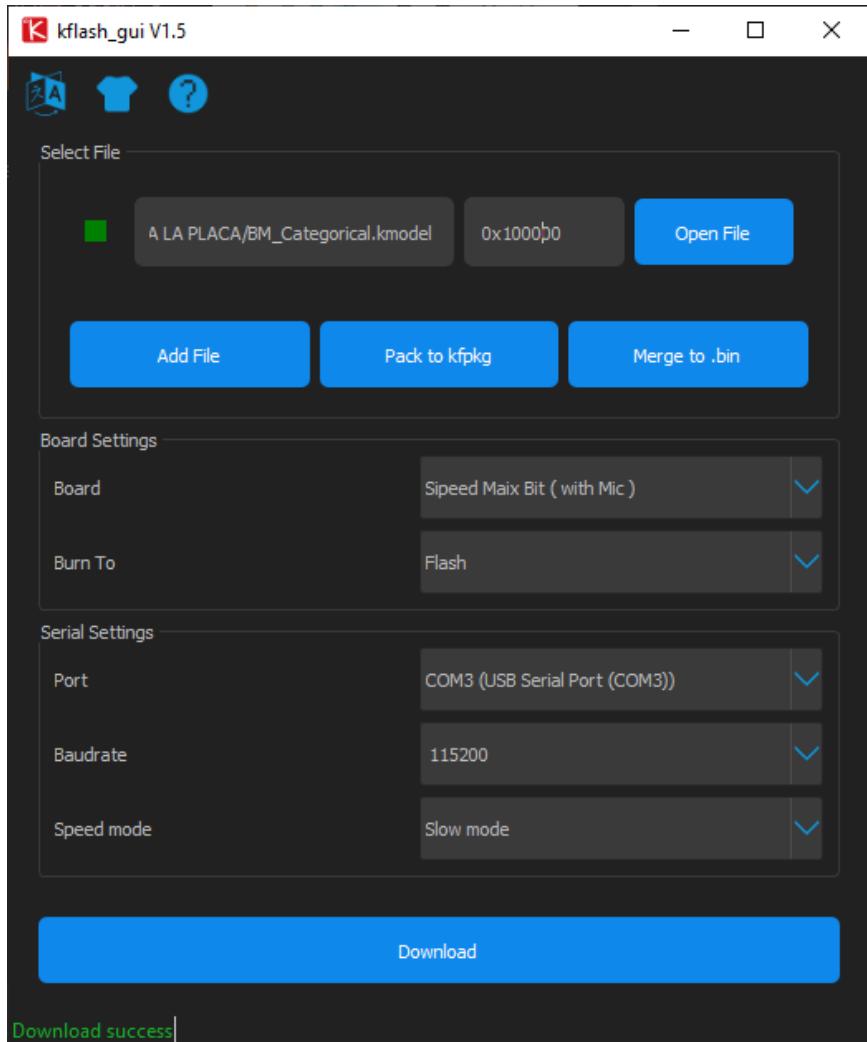
**Figura 84.** Placa con firmware cargado

Una vez tenemos el firmware correctamente cargado ya podemos empezar a manejar la placa con el uso del programa Putty, el cual explicaremos en la sección [4.3.4](#) junto a los scripts necesarios para hacer funcionar al modelo.

Para que el modelo se encuentre también en la memoria interna de la placa debemos cargarlo de la misma forma que cargamos el firmware, pero esta vez indicando una dirección de memoria que será a la que accedamos desde el script de MicroPython.

En el programa K-Flash GUI dejamos la misma información que teníamos para cargar el firmware cambiando únicamente el archivo a cargar en la placa, que en nuestro caso es el modelo entrenado con extensión **.kmodel**. En nuestro proyecto cargamos la red en la dirección 0x100000, que será a la que accedamos posteriormente para hacer funcionar a nuestro modelo.

Al igual que al cargar el firmware, nos encontraremos con los dos leds de la placa encendidos y otra barra de carga, indicando que el proceso de carga se está realizando de forma correcta sobre el dispositivo.



**Figura 85.** Configuración para cargar el modelo a la placa

Al terminar de cargar el modelo la placa estará completamente lista para ser usada, teniendo únicamente que generar el archivo de MicroPython que use las funciones internas de la placa y nos ayude a cargar y ejecutar el modelo sobre las imágenes captadas por el sensor óptico.

#### 4.3.4 Ejecución del modelo

En este subcapítulo explicaremos los últimos pasos a dar para conseguir ejecutar el modelo creado en la placa, de forma que seamos capaces de establecer conexión entre los componentes íntegros y nuestro modelo personalizado.

Es importante saber que la placa recibirá las imágenes captadas por la cámara cada *frame*, siendo de vital importancia controlar el tiempo de captura y la calidad de las imágenes.

La ejecución del modelo consume un cierto tiempo necesario para clasificar las imágenes, teniendo que esperar este tiempo antes de volver a generar otra imagen para evitar superposición y errores en la clasificación.

Para comenzar con la ejecución del modelo debemos crear un pequeño *script* de MicroPython, en el cual incluiremos las diferentes funciones y las diferentes instrucciones para hacer funcionar todos los componentes de la placa correctamente.

En nuestro caso hemos usado la interfaz de [MaixPy IDE](#) para elaborar las instrucciones usando el predictor, el cual nos ha ayudado mucho a completar las diferentes instrucciones.

Para hacer una explicación un poco más detallada vamos a ir paso a paso con las diferentes instrucciones usadas y las diferentes funciones usadas en el proceso. Como base del código hemos usado un [script prefabricado](#) donde se usaban estas mismas instrucciones de manera similar.

En primer lugar, debíamos añadir las diferentes librerías necesarias para el manejo de los componentes de la placa tales como los leds, la cámara o la pantalla LCD. Dentro de estas librerías también nos encontramos con las necesarias para tener un control del tiempo y por supuesto la necesaria para ejecutar nuestro modelo.

```
import sensor, image, lcd, time
import KPU as kpu
from fpioa_manager import fm
from board import board_info
from Maix import GPIO
```

**Figura 86.** Imports utilizados en el script de MicroPython

El manejo de los pines de salida debe realizarse con el uso de las librerías [FPIOA Manager](#) y [GPIO](#), las cuales incluyen las características necesarias para crear las referencias a los pines y poder usarlos.

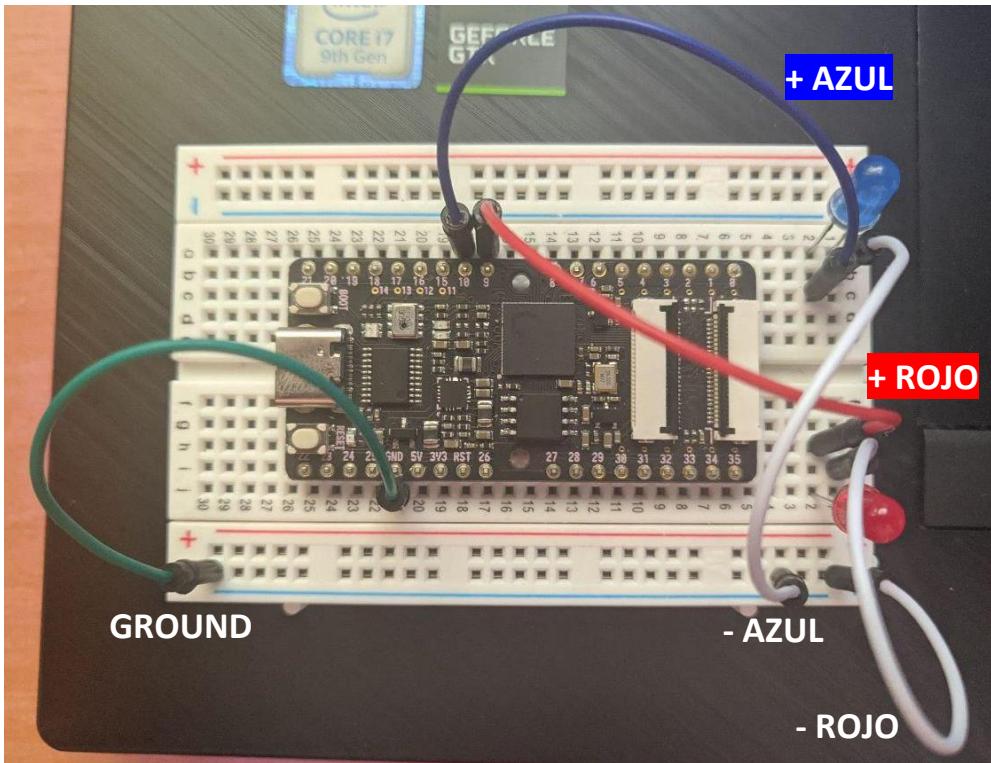
Algo importante para tener en cuenta es que, aunque la placa disponga de 35 pines a su alrededor tan solo 8 son de propósito general, quedando el resto inservibles para usar nuestros leds externos.

```
fm.register(board_info.PIN9, fm.fpioa.GPIO0)
led_perro=GPIO(GPIO.GPIO0,GPIO.OUT)
fm.register(board_info.PIN10, fm.fpioa.GPIO1)
led_gato=GPIO(GPIO.GPIO1,GPIO.OUT)
```

**Figura 87.** Inicialización de los pines de salida

Los pines que vamos a usar en nuestro caso van a ser los pines 9 y 10, los cuales debemos registrar dentro del “*manager*” y, posteriormente, clasificarlos como pines de salida. Esto indicará que estos pines darán una corriente eléctrica o no dependiendo del valor que tenga asociado en un cierto tiempo (0 o 1), consiguiendo así encender los leds.

El montaje de los leds en la placa es muy sencillo y lo tenemos indicado por los colores de los cables, asociando al pin 9 el color rojo y al 10 el color azul. Lo único que debemos hacer es hacer pasar la corriente dada por los leds llevando la salida del led al pin de tierra, el cual es común para ambos leds (indicado con el cable blanco).



**Figura 88.** Esquema de la placa con los leds

La función que vamos a usar para activar los leds va a recibir el índice de la clase con mayor porcentaje de pertenencia y va a activar el led correspondiente a esta. La forma de activarlo es indicando como valor de salida el 1.

```
def lightUpLed(classIdx):

    led_perro.value(0)
    led_gato.value(0)

    if (classIdx == 0):
        led_perro.value(1)
        # LED PERRO = ROJO
    if (classIdx == 1):
        led_gato.value(1)
        # LED GATO = AZUL
```

**Figura 89.** Función para el manejo de los leds

Con el apartado de los leds inicializado podemos comenzar con la configuración del sensor para que las imágenes capturadas sean de la máxima calidad posible y que pueda cuadrar con las imágenes de entrada del modelo.

Esta [librería](#) nos permite modificar múltiples valores como el formato de los píxeles o el tamaño del *frame*. El que más importancia tiene para nosotros es el tamaño de la imagen tomada, ya que debe ser igual al del tamaño de la imagen de entrada de nuestro modelo.

```

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.set_windowing((224, 224))
sensor.set_vflip(0)
sensor.set_hmirror(False)
sensor.run(1)

```

**Figura 90.** Inicialización del sensor

Para el [LCD](#) únicamente debemos inicializarlo con una cierta frecuencia para poder utilizarlo. Sin embargo, para dar un poco más de personalización, hemos añadido un pequeño texto que indica que pertenece a este TFG.

La forma que tenemos de pintar en la pantalla es indicando las coordenadas dentro de la pantalla junto al texto que queramos poner indicado como un *String*. La coordenada 0,0 empieza en la esquina superior izquierda.

```

lcd.init(freq=15000000)
lcd.clear()
lcd.draw_string(30, 96, "Clasificador entre perros y gatos")
lcd.draw_string(60, 112, "TFG Daniel Riveros Garcia")
lcd.draw_string(70, 128, "ING. INFORMATICA HUELVA")

```

**Figura 91.** Inicialización de nuestro LCD

El siguiente paso es preparar el script para ejecutar nuestro modelo. La librería encargada del manejo de los modelos es la denominada como [KPU](#), la cual incluye un apartado dirigido para clasificadores y otro dirigido a encontrar objetos en la imagen usando la tecnología [yolo](#).

Nosotros iremos por la rama dirigida a los clasificadores para la cual nos hace falta cargar el modelo (desde SD o desde la memoria interna) y tener guardadas las clases que vamos a usar en nuestro clasificador.

```

labels = ['Gato', 'Perro']
# Desde SD
# task = kpu.load('/sd/model.kmodel')
# Desde memoria interna
task = kpu.load(0x100000)

```

**Figura 92.** Creación de etiquetas y carga del modelo

Con todas las variables inicializadas podemos crear el bucle infinito el cual se encargará de recoger la imagen del sensor, pasarlala por el modelo y clasificarla hacia alguna de las dos clases, mostrando el resultado por pantalla y encendiendo el led correspondiente.

```

img = sensor.snapshot()
clock.tick()
# Tiempo de espera para bajar los fps (aprox 12 imágenes por segundo)
time.sleep(1/12)
# Generamos un array de probabilidades con el modelo y la imagen
fmap = kpu.forward(task, img)
fps = clock.fps()

# Obtenemos la máxima probabilidad
plist = fmap[:,]
pmax = max(plist)

```

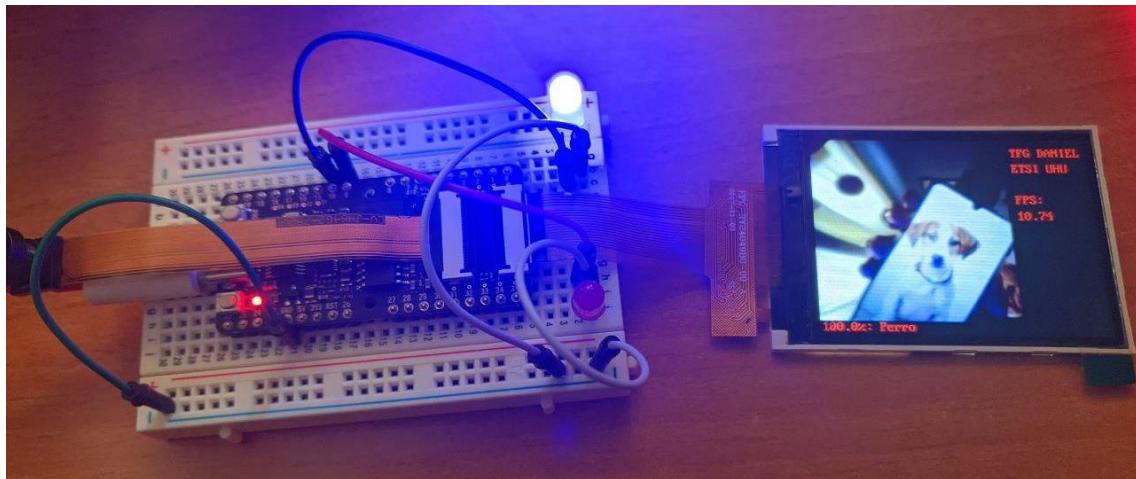
**Figura 93.** Clasificación de la imagen con nuestro modelo

Usando la librería KPU podemos obtener el array de probabilidades de pertenencia que tiene la imagen con cada una de las clases. Esto se consigue gracias a tener una neurona de salida por cada clase y usando la función de activación *softmax*, la cual es la compatible con esta instrucción.

Una vez tenemos el array de probabilidades solo nos falta obtener el índice mayor, indicando si es perro o gato en ese orden, indicando la clase resultante junto con el porcentaje de pertenencia. El resultado se mostrará en la pantalla junto a la imagen y un pequeño texto indicando el TFG y los *fps* actuales.

En la figura 93 podemos ver que añadimos un pequeño *sleep* para lograr aproximadamente 12 imágenes por segundo, ya que, si los *fps* fueran muy elevados la clasificación podría tener interferencias y no ser muy precisa.

El resto del código se encuentra en nuestro [repositorio del TFG](#), donde podemos ver el resto de instrucciones usadas para mostrar el resultado y la forma de tratar los porcentajes obtenidos.



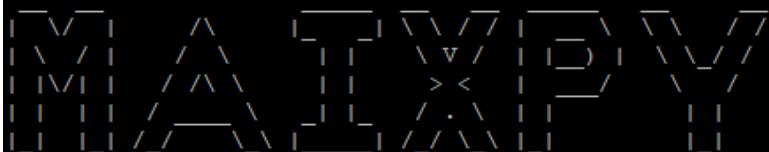
**Figura 94.** Resultados finales de la ejecución

#### 4.3.5 Ejecución automática del modelo

Todos los ficheros cargados desde la aplicación quedan guardados en la memoria interna de la placa, evitando tener que cargar todos los ficheros necesarios cada vez que queramos utilizarla.

De igual manera, cada vez que enchufamos la placa a la corriente el sistema se queda en pausa esperando la introducción de instrucciones. En ese momento nosotros podemos introducir todo el código que queramos, como, por ejemplo, el visto en el apartado anterior.

```
[MAIXPY] P110:freq:832000000  
[MAIXPY] P111:freq:398666666  
[MAIXPY] P112:freq:450666666  
[MAIXPY] cpu:freq:416000000  
[MAIXPY] kpu:freq:398666666  
[MAIXPY] Flash:0xc8:0x17  
open second core...  
gc heap=0x800d46d0-0x801546d0 (524288)  
[MaixPy] init end
```



Official Site : <https://www.sipeed.com>  
Wiki : <https://maixpy.sipeed.com>

**Figura 95.** Interfaz inicial de la placa desde Putty

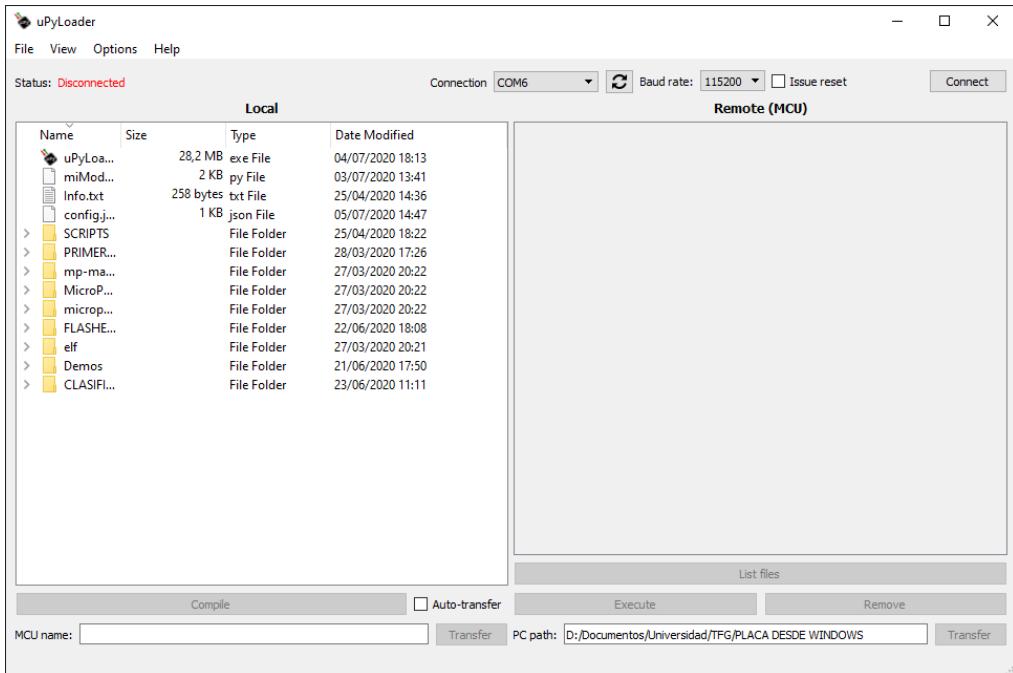
Nuestro proyecto, al tratarse de un sistema tan específico, podemos automatizarlo de forma que sea el programa principal que este cargado en la placa, el cual es el que se ejecutará al iniciarse la placa por primera vez.

Este concepto tiene varias ventajas, como, por ejemplo, la inicialización automática de la aplicación en sistemas que incluyan esta tecnología sin la necesidad de un operario de mantenimiento.

En nuestro caso nos servirá para demostrar el funcionamiento del dispositivo de forma individual, sin la necesidad de un ordenador que lo controle de forma remota o de personal especializado en el manejo de dispositivos de la misma índole.

Para poder cargar nuestro script personalizado debemos entender de qué forma funciona la placa en el interior. Este funcionamiento es bastante sencillo de entender y consiste en ejecutar dos ficheros presentes en el firmware de la placa, los cuales indican funciones diferentes. El primero se ejecuta nada más conectar la corriente como inicializador principal mientras que el segundo se ejecuta como script principal tras la inicialización.

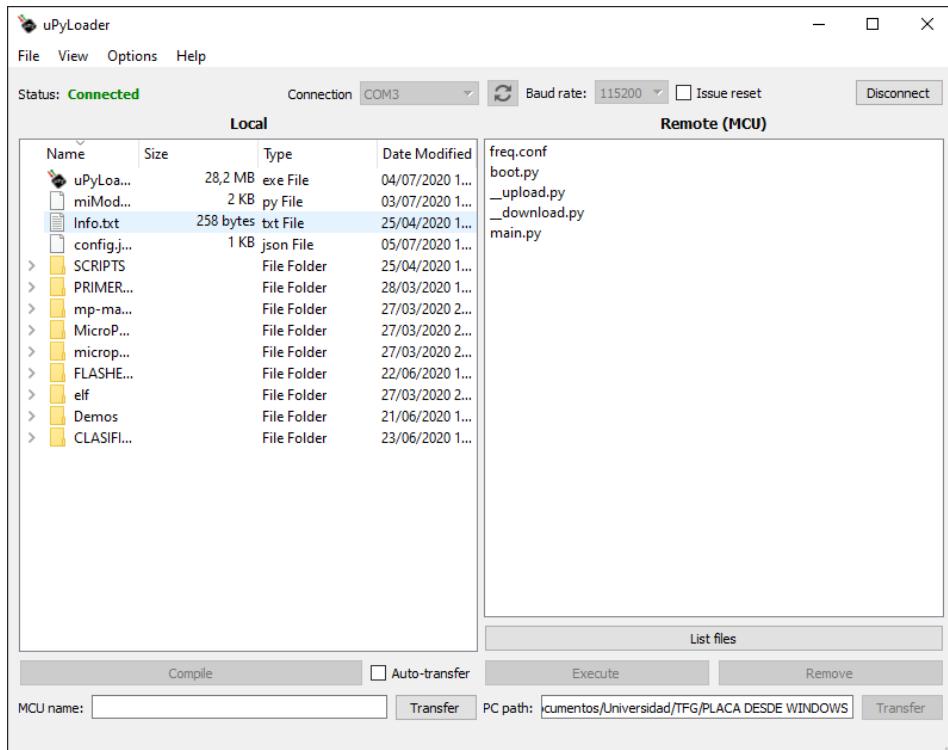
Estos ficheros están ocultos y para poder verlos necesitamos usar el programa [uPyLoader](#), el cual nos permite acceder a los ficheros internos de la placa y poder modificarlos a nuestro antojo.



**Figura 96.** Interfaz principal de la ampliación uPyLoader

Esta aplicación se encargará de crear los archivos necesarios para cargar y modificar los ficheros anteriormente explicados, los cuales nos servirán para ejecutar nuestro propio script en lugar de los scripts por defecto.

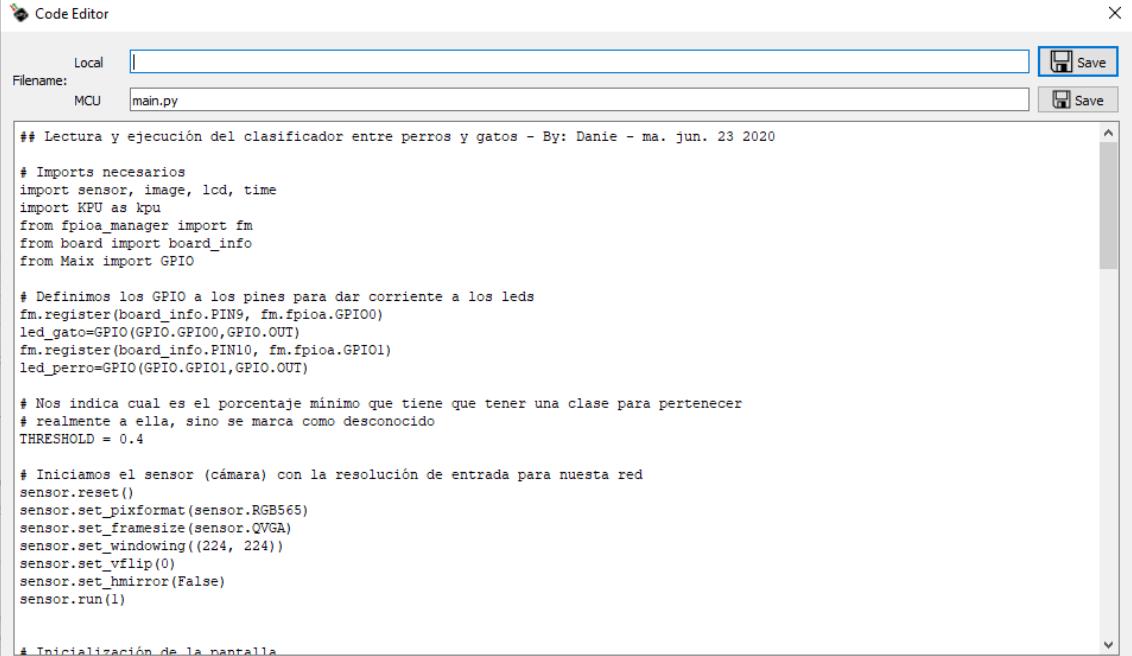
Una vez la placa está conectada podemos ver los archivos internos que ejecuta la placa al iniciarse, siendo el fichero `boot.py` el inicializador y el fichero `main.py` el script principal de nuestro dispositivo.



**Figura 97.** Interfaz con el dispositivo conectado

A nosotros nos interesa ejecutar nuestro script como ejecución principal dentro de nuestra placa, teniendo que cambiar el código presente en el script `main.py` por el presente en el script creado en el capítulo [4.3.4](#).

Este paso se puede realizar directamente desde el editor del programa, el cual nos permite modificar el código presente dentro de cualquiera de los scripts presentes de forma sencilla, teniendo únicamente que reemplazar el código original por el nuestro.



The screenshot shows a software interface titled "Code Editor". At the top, there are tabs for "Local" and "MCU", with "MCU" selected. Below the tabs, the file name is listed as "main.py". There are two "Save" buttons: one for the local copy and one for the MCU copy. The main window displays Python code for a dog-cat classifier. The code includes imports for sensor, image, lcd, time, KPU, and fpioa\_manager. It defines GPIO pins for LEDs and initializes the sensor with specific parameters like resolution and frame size. A threshold variable is set to 0.4. The code ends with a note about initializing the screen.

```
## Lectura y ejecución del clasificador entre perros y gatos - By: Danie - ma. jun. 23 2020

# Imports necesarios
import sensor, image, lcd, time
import KPU as kpu
from fpioa_manager import fm
from board import board_info
from Maix import GPIO

# Definimos los GPIO a los pines para dar corriente a los leds
fm.register(board_info.PIN9, fm.fpioa.GPIO0)
led_gato=GPIO(GPIO.GPIO0,GPIO.OUT)
fm.register(board_info.PIN10, fm.fpioa.GPIO1)
led_perro=GPIO(GPIO.GPIO1,GPIO.OUT)

# Nos indica cual es el porcentaje mínimo que tiene que tener una clase para pertenecer
# realmente a ella, sino se marca como desconocido
THRESHOLD = 0.4

# Iniciamos el sensor (cámara) con la resolución de entrada para nuestra red
sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.set_windowing((224, 224))
sensor.set_vflip(0)
sensor.set_hmirror(False)
sensor.run(1)

# Inicialización de la pantalla
```

**Figura 98.** Editor de textos de uPyLoader

Una vez modificado este fichero la placa estará lista para ejecutar nuestro script nada más recibir la corriente eléctrica sin necesidad de ningún ordenador para hacerla funcionar.

Con esta implementación tenemos nuestro clasificador perfectamente funcional en cualquier situación que se nos pueda presentar, siendo fácil cambiarlo de posición o incluso modificar los ficheros en su interior para implementar otro tipo de red sin necesidad de muchos cambios.

Esto nos permite conseguir un sistema móvil y potente, el cual es capaz de realizar multitud de tareas con tan solo la necesidad de estar conectado a una red eléctrica pequeña, la cual puede ser prevista desde un ordenador, una toma de pared o incluso una *PowerBank*.

---

## Capítulo 5

# Conclusiones y Trabajo Futuro

---

A lo largo de nuestro proyecto hemos hecho un repaso detallado de las diferentes tecnologías usadas a lo largo del trabajo de experimentación, consiguiendo afianzar los diferentes conceptos adquiridos en la carrera y aprendiendo otros tantos que se quedan fuera de los límites de esta.

Las redes neuronales convolucionales son y han sido una revolución en el mundo de la inteligencia artificial. Desde sus inicios ya se comenzó a usar en diferentes proyectos en todo el mundo dedicados cada uno a categorías diferentes.

El uso de este tipo de tecnología dota a este trabajo de actualidad y de interés ya que, puede servir para su implementación en trabajos futuros o incluso en su inclusión en el ámbito profesional.

Ser capaces de controlar una red tan potente desde un dispositivo con unas dimensiones tan reducidas nos demuestran la gran movilidad que ha adquirido la informática en estos últimos años y como hemos pasado de grandes habitaciones que formaban un gran ordenador hasta los móviles que usamos diariamente y nos caben en la palma de la mano.

El estudio y la implementación de una red personalizada sobre este tipo de placas nos ha servido para elaborar un repositorio “guía” para futuras implementaciones de otros usuarios interesados en la materia, y es que, tras el estudio de todos los componentes necesarios para su implementación, nos hemos dado cuenta de la falta de documentación, la cual queremos suplir con este trabajo.

Los datos sobre la implementación o incluso la ejecución del modelo en la propia placa venían sin explicar y dedicados a redes prefabricadas sobre un modelo conocido (como por ejemplo [MobileNet](#)), pero no sobre un modelo hecho desde 0, algo que aporta flexibilidad y libertad para los desarrolladores de este tipo de software.

Otro punto fuerte para tener en cuenta son las diferentes implementaciones que hemos llevado al sistema operativo Windows, el cual es el más usado en todo el mundo y va a hacer más accesible llevar este conocimiento a práctica por parte de más usuarios, consiguiendo, en consecuencia, más proyectos y más avances sobre este tipo de tecnología.

Este trabajo nos ha servido para conocer de primera mano las diferentes limitaciones que tienen este tipo de software junto a los diferentes modelos existentes y los diferentes usos que se les da a cada uno, especializándose cada uno en una función diferente.

Nuestro modelo funciona como clasificador binario, el cual es capaz de diferenciar fácilmente entre dos clases sencillas. La implementación de este modelo sobre la placa nos demuestra que la introducción de modelos clasificadores sobre este tipo de hardware es posible con un poco de conocimiento sobre esta materia.

Como trabajo final de grado nos ha resultado muy satisfactorio conseguir el funcionamiento de nuestro modelo sobre el dispositivo, dejando demostrados todos los conceptos explicados a lo largo del proyecto.

Como trabajo personal, la incorporación de estas redes tan potentes tras la modificación de varios ficheros y tras la adaptación de varios recursos nos hace pensar que ha valido la pena tanto esfuerzo. Esta adaptación va a poder usada por otros usuarios con el mismo interés que el nuestro, los cuales podrán ahorrarse muchos pasos en falso siguiendo la guía [anexa](#) en esta memoria.

Demostrada la funcionalidad de nuestro clasificador sobre una placa tan sencilla, tenemos una base muy sólida sobre lo que se puede conseguir siguiendo un modelo parecido sobre otro tipo de placas más complejas y precisas.

Nuestra placa, por ejemplo, únicamente trabaja de forma local sin tener conexión a internet, algo muy importante para proyectos más profesionales o incluso para su inclusión en situaciones reales.

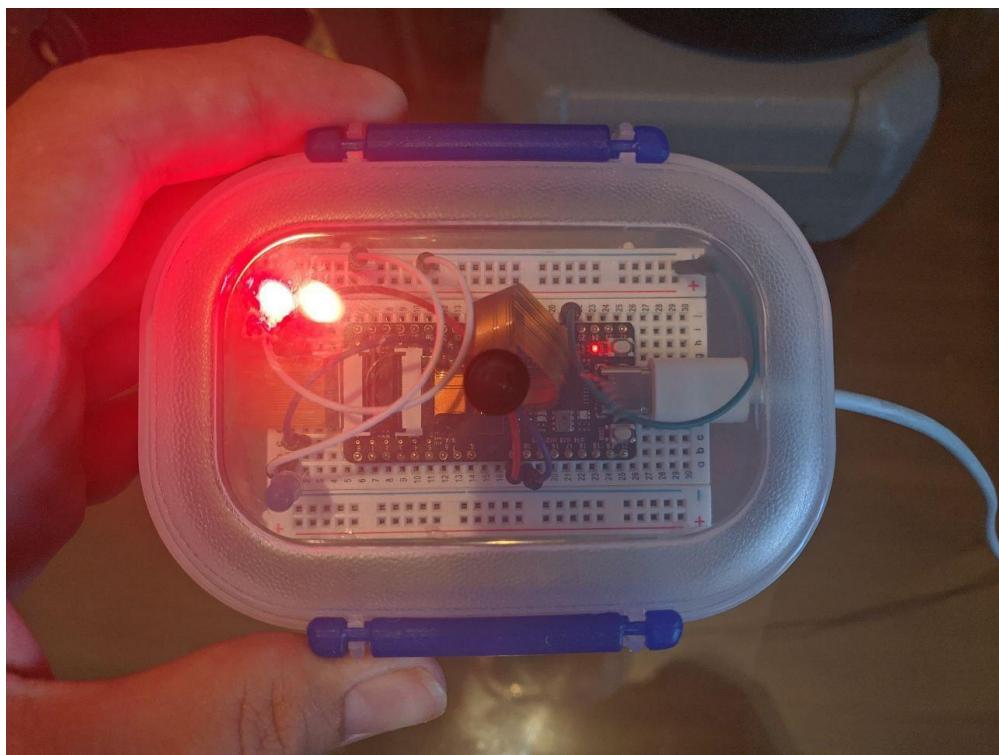
Existen modelos de esta misma marca más profesionales que incluyen wifi junto a un poco más de memoria, permitiendo así la inclusión de modelos más complejos que pueden ser usados de forma remota e incluso modelos que pueden ser autoentrenados con imágenes captadas con la cámara de la placa.

A partir de esta base y guía existen multitud de vertientes que seguir, desde mejorar nuestro modelo para conseguir mayor precisión, cambiar las imágenes clasificando entre otro tipo de clases o incluso incluir más opciones al clasificador.

De igual forma, sea cual sea el objetivo siguiente, el trabajo realizado no será en vano, ya que, al haber construido una buena primera implementación y habiendo estudiado todos los conceptos necesarios, va a ser mucho más fácil comenzar un proyecto nuevo ahorrándonos cometer los mismos fallos que hemos podido cometer a lo largo de este trabajo.

Una vez expuesto estos conceptos y al haber conseguido todos los objetivos propuestos en la introducción podemos dar por finalizado este proyecto tras meses de intensas pruebas y corrección de fallos con un resultado muy positivo, dejando una buena primera guía para futuros trabajos parecidos sobre conceptos parecidos.

Podemos dar por finalizado la realización de nuestro TFG.



**Figura 99.** Modelo final de la placa. Vista Superior



**Figura 100.** Modelo final de la placa. Vista Inferior

---

---

## Capítulo 6

## Bibliografía

---

- [Zhang et al, 2018] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin y Jian Sun (2018). *ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices.* [PDF](#)
- [McCulloch y Pitts, 1943] Warren S. McCulloch y Walter H. Pitts (1943). *A logical Calculus Of The Ideas Immanent In Nervous Activity.* [PDF](#)
- [Hebb, 1949] D. O. Hebb (1949). *A Neuropsychological Theory.* [PDF](#)
- [Rosenblatt, 1958] F. Rosenblatt (1958). *The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain.* [PDF](#)
- [Rummelhart et al, 1986] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams (1986). *Learning Representations By Back-Propagation Errors.* [PDF](#)
- [Yann LeCun et al, 1989] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel (1989). *Backpropagation Applied to Handwritten Zip Code Recognition.* [PDF](#)
- [Hinton et al, 2006] Geoffrey E. Hinton, Simon Osindero y Yee-Whye Teh (2006). *A Fast Learning Algorithm for Deep Belief Nets.* [PDF](#)
- [Bengio et al, 2007] Yoshua Bengio, Pascal Lamblin, Dan Popovici y Hugo Larochelle (2007). *Greedy Layer-Wise Training of Deep Networks.* [PDF](#)
- [Ranzato et al, 2007] Marc'Aurelio Ranzato, Y-Lan Boureau y Yann LeCun (2007). *Sparse Feature Learning for Deep Belief Networks.* [PDF](#)
- [Jianxin Wu, 2017] Jianxin Wu (2017). *Introduction to Convolutional Neural Networks.* [PDF](#)
- [Jay Kuo 2016] C.-C. Jay Kuo (2016). *Understanding Convolutional Neural Networks with Mathematical Model.* [PDF](#)
- [Kaiming He et al, 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* [PDF](#)

- [Dominik Scherer et al, 2010]. Dominik Scherer, Andreas Müller y Sven Behnke (2010). *Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition*. [PDF](#)
- [Andrew Waterman et al, 2011] Andrew Waterman, Yunsup Lee, David A. Patterson y Krste Asanovic (2011). *The Risc-V Instruction Set Manual, Volume I. Base User-Level ISA*. [PDF](#)
- [He et al, 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. [PDF](#)
- [Simonyan y Zisserman, 2014] Karen Simonyan y Andrew Zisserman (2014). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. [PDF](#)
- [Kiefer y Wolfowitz, 1952] J. Kiefer y J. Wolfowitz (1952). *Stochastic Estimation of the Maximum of a Regression Function*. [PDF](#)
- [Kingma y Ba, 2017] Diederik P. Kingma y Jimmy Ba (2017). *Adam: A Method for Stochastic Optimization*. [PDF](#)
- [Ozkaya y Seyfi, 2019] Umut Ozkaya y Levent Seyfi (2019). *Fine-Tuning Models Comparisons on Garbage Classification for Recyclability*. [PDF](#)
- (TyN Magazine) *20 ejemplos de aprendizaje automático utilizado en la experiencia del cliente* (2019). [ENLACE](#)
- (Intel) *Cómo las empresas inteligentes avanzan con el aprendizaje automático*. [ENLACE](#)
- (Atria Innovation) *Deep Learning y sus muchas aplicaciones* (2019). [ENLACE](#)
- (Byte) *Deep Learning, la gran oportunidad para la automoción, el retail o las finanzas* (2017). [ENLACE](#)
- (Patricio Loncomilla, 2016) *Deep learning: Redes convolucionales* (2016). [ENLACE](#)
- (IBM) *Redes neuronales convolucionales* (2016). [ENLACE](#)
- (Diego Calvo) *Red Neuronal Convolucional CNN* (2017). [ENLACE](#)
- (Medium) *Una Introducción Completa A Redes Neuronales Con Python y Tensorflow 2.0* (2019). [ENLACE](#)
- (Medium) *Introducción al deep learning parte 2: Redes Neuronales Convolucionales* (2019). [ENLACE](#)

- (Aprende Machine Learning) *¿Cómo funcionan las Convolutional Neural Networks? Visión por Ordenador* (2018). [ENLACE](#)
- (CS231n) *Convolutional Neural Networks for Visual Recognition*. [ENLACE](#)
- (Towards Data Science) *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way* (2018). [ENLACE](#)
- (SuperDataScience) *The Ultimate Guide to Convolutional Neural Networks (CNN)* (2018). [ENLACE](#)
- (Health Big Data) *Redes Neuronales Convolucionales*. [ENLACE](#)
- (UAB) *Se crea la Red-RISCV para impulsar el desarrollo de hardware de código abierto*. [ENLACE](#)
- (HardwareLibre) *RISC-V, el primer procesador libre para nuestros proyectos*. [ENLACE](#)
- (RISC-V) Página oficial. [ENLACE](#)
- (Wikipedia) Risc-V. [ENLACE](#)
- (SeeedStudio) *Get Started with K210: Hardware and Programming Environment* (2019). [ENLACE](#)
- (SiFive) *Página Oficial*. [ENLACE](#)
- (Pic Microcontroller). *OPEN SOURCE MEETS HARDWARE: OPEN PROCESSOR CORE* (2018). [ENLACE](#)
- (Oracle) *¿Qué es IoT?* [ENLACE](#)
- (Deloitte) *IoT - Internet Of Things*. [ENLACE](#)
- (Wikipedia) *MicroPython*. [ENLACE](#)
- (MicroPython) *Página Oficial*. [ENLACE](#)
- (Ieec) *Ingeniería de los Sistemas Embebidos*. [ENLACE](#)
- (El Financiero) *¿Sabes qué es un sistema embebido?* [ENLACE](#)
- (Incibe-cert) *Introducción a los sistemas embebidos* (2018). [ENLACE](#)

- (SemanticWebBuilder) *Sistemas Embebidos: Innovando hacia los Sistemas Inteligentes*. [ENLACE](#)
- (Medium) *Xavier and He Normal (He-et-al) Initialization* (2018). [ENLACE](#)
- (Towards Data Science) *Stochastic Gradient Descent — Clearly Explained !!* (2019). [ENLACE](#)
- (Machine Learning Mastery) *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning* (2017). [ENLACE](#)
- (Wikipedia) *Sensitivity and specificity*. [ENLACE](#)
- (TensorFlow) *Tensorflow Lite converter*. [ENLACE](#)
- (Rast Github) *Confusion Matrix*. [ENLACE](#)
- (Scikit Learn) *Confusion matrix*. [ENLACE](#)
- (Towards Data Science) *Multi-class Classification: Extracting Performance Metrics From The Confusion Matrix* (2019). [ENLACE](#)
- (Principles of Deep Learning) *A TUTORIAL ON GLOBAL AVERAGE POOLING*. [ENLACE](#)
- (Aprende Machine Learning) *Principales Algoritmos usados en Machine Learning* (2017). [ENLACE](#)
- (Aprende Machine Learning) *Breve Historia de las Redes Neuronales Artificiales* (2018). [ENLACE](#)
- (DataHack) *HISTORIA DEL DEEP LEARNING ETAPAS* (2019). [ENLACE](#)
- (En mi Local Funciona) *Deep Learning básico con Keras (Parte 1)* (2018). [ENLACE](#)
- (En mi Local Funciona) *Deep Learning básico con Keras (Parte 2): Convolutional Nets* (2018). [ENLACE](#)
- (En mi Local Funciona) *Deep Learning básico con Keras (Parte 3): VGG* (2018). [ENLACE](#)
- (Machine Learning Mastery) *How to Classify Photos of Dogs and Cats (with 97% accuracy)* (2019). [ENLACE](#)

- (Hackster) *Object Detection with Sipeed MaiX Boards (Kendryte K210)* (2019).  
[ENLACE](#)
- (Mouser) *Información sobre la placa.* [ENLACE](#) [ENLACE2](#)
- (Seeed Studio) *Información sobre la placa, página oficial.* [ENLACE](#)
- (TeslaBem) *Qué es MicroPython* (2016). [ENLACE](#)
- (Sipeed Blog) *¡Entrene, convierta, ejecute MobileNet en Sipeed MaixPy y MaixDuino!* (*Traducido de chino por Google*). [ENLACE](#)
- (Machine Learning Mastery) *How to Configure Image Data Augmentation in Keras* (2019). [ENLACE](#)
- (MaixPy Sipeed) Official Documentation. [ENLACE](#).
- (Hack A Day) *Object detector - MobileNet and YOLOv2 on MAixPy.* [ENLACE](#)
- (Element14) *Sipeed Maixduino - MaixPy Image Classification.* [ENLACE](#)

---

# Capítulo 7

## Índice de figuras

---

1. [https://obrazki.elektroda.pl/7190683900\\_1560700459\\_bighthumb.jpg](https://obrazki.elektroda.pl/7190683900_1560700459_bighthumb.jpg)
2. [https://miro.medium.com/max/600/1\\*c4EsT8qTEur94V8ZsIN1xQ.png](https://miro.medium.com/max/600/1*c4EsT8qTEur94V8ZsIN1xQ.png)
3. <https://i.pinimg.com/originals/40/ec/d8/40ecd87c7f41c31a58ad21fb0796c5b2.png>
4. <https://www.cs.us.es/~fsancho/images/2019-12/machine-learning-types.jpg>
5. <https://i.pinimg.com/originals/e7/77/31/e77731ce84921b2678f5c31830c7241c.png>
6. <https://raw.githubusercontent.com/SeeedDocument/Outsourcing/master/Sipeed%20Pic/Sipeed-Bit-intro.jpg>
7. Captura página oficial de Maix Bit
8. Tema perceptrones asignatura AA
9. <https://i.stack.imgur.com/7Ui1C.png>
10. [https://miro.medium.com/max/2550/0\\*sq5-Moj6ipTEMW21.png](https://miro.medium.com/max/2550/0*sq5-Moj6ipTEMW21.png)
11. <https://www.atriainnovation.com/wp-content/uploads/2019/10/deteccion.jpg>
12. [https://cdn-images-1.medium.com/freeze/max/1000/1\\*FQormTY9aoWB\\_sioKFQZw.png?q=20](https://cdn-images-1.medium.com/freeze/max/1000/1*FQormTY9aoWB_sioKFQZw.png?q=20)
13. <https://www.top10motor.com/wp-content/uploads/2018/07/coche-autonomo-4.jpg>
14. [https://raw.githubusercontent.com/ssusnic/Machine-Learning-Flappy-Bird/master/screenshots/flappy\\_06.png](https://raw.githubusercontent.com/ssusnic/Machine-Learning-Flappy-Bird/master/screenshots/flappy_06.png)
15. [https://i.blogs.es/de8d41/china/450\\_1000.jpg](https://i.blogs.es/de8d41/china/450_1000.jpg)
16. <https://365datascience.com/wp-content/uploads/2017/11/Backpropagation.jpg>

## 17. PDF RECURSOS

18. [https://la.mathworks.com/solutions/deep-learning/convolutional-neural-network/\\_jcr\\_content/mainParsys/band\\_copy\\_copy\\_14735\\_1026954091/mainParsys/columns\\_1606542234\\_c/2/image.adapt.full.low.jpg/1583131829796.jpg](https://la.mathworks.com/solutions/deep-learning/convolutional-neural-network/_jcr_content/mainParsys/band_copy_copy_14735_1026954091/mainParsys/columns_1606542234_c/2/image.adapt.full.low.jpg/1583131829796.jpg)
19. <https://torres.ai/wp-content/uploads/2018/06/Picture.4.1.png>
20. <https://setosa.io/ev/image-kernels/>
21. <https://cs.nju.edu.cn/wujx/paper/CNN.pdf>
22. [https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35\\_blog\\_image\\_12.png](https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35_blog_image_12.png)
23. [https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35\\_blog\\_image\\_16.png](https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35_blog_image_16.png)
24. <https://i1.wp.com/www.aprendemachinelearning.com/wp-content/uploads/2018/11/CNN-04.png>
25. [https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35\\_blog\\_image\\_20.png](https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35_blog_image_20.png)
26. <https://i2.wp.com/principlesofdeeplearning.com/wp-content/uploads/2018/07/poolingmax.png?resize=600%2C264>
27. [https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35\\_blog\\_image\\_24.png](https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35_blog_image_24.png)
28. [https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35\\_blog\\_image\\_32.png](https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/35_blog_image_32.png)
29. [https://miro.medium.com/max/1390/1\\*gWbpNBY07DhTSt8nUwlNLg.jpeg](https://miro.medium.com/max/1390/1*gWbpNBY07DhTSt8nUwlNLg.jpeg)
30. <https://pic-microcontroller.com/wp-content/uploads/2017/11/SiFive-FE310.jpg>
31. <https://conexiona.com/wp-content/uploads/2020/02/internet-of-things.png>
32. <https://www.tensorflow.org/site-assets/images/project-logos/tensorflow-lite-logo-social.png>

**33.** <https://agelectro904833371.files.wordpress.com/2019/08/micropython-logo.jpg>

**34.** [https://www.incibe-cert.es/sites/default/files/blog/sistemas\\_embedidos/grafica2es.png](https://www.incibe-cert.es/sites/default/files/blog/sistemas_embedidos/grafica2es.png)

**35.** <https://www.aa1car.com/library/bywire2.jpg>

**36.** <https://kakalabblog.wordpress.com/2017/07/27/implementing-cnn-in-tensorflow/>

## **37. DATASETS**

**38.** [https://miro.medium.com/max/850/1\\*ae1tW5ngf1zhPRyh7aaM1Q.png](https://miro.medium.com/max/850/1*ae1tW5ngf1zhPRyh7aaM1Q.png)

**39.** [https://www.researchgate.net/profile/Max\\_Ferguson/publication/322512435/figure/fig3/AS:697390994567179@1543282378794/Fig-A1-The-standard-VGG-16-network-architecture-as-proposed-in-32-Note-that-only.png](https://www.researchgate.net/profile/Max_Ferguson/publication/322512435/figure/fig3/AS:697390994567179@1543282378794/Fig-A1-The-standard-VGG-16-network-architecture-as-proposed-in-32-Note-that-only.png)

**40.** [https://cdn-images-1.medium.com/max/1600/0\\*fU8XFt-NCMZGAWND.](https://cdn-images-1.medium.com/max/1600/0*fU8XFt-NCMZGAWND.)

**41.** <https://i.stack.imgur.com/wEfbW.png>

**42.** <https://blog.paperspace.com/content/images/2018/06/adam.png>

**43.** <https://3qepr26caki16dnhd19sv6by6v-wpengine.netdna-ssl.com/wp-content/uploads/2017/05/Comparison-of-Adam-to-Other-Optimization-Algorithms-Training-a-Multilayer-Perceptron.png>

## **44. Captura de Pantalla**

**45.** [https://en.wikipedia.org/wiki/Sensitivity\\_and\\_specificity#/media/File:HighSensitivity\\_LowSpecificity\\_1401x1050.png](https://en.wikipedia.org/wiki/Sensitivity_and_specificity#/media/File:HighSensitivity_LowSpecificity_1401x1050.png)

**46.** [https://en.wikipedia.org/wiki/Sensitivity\\_and\\_specificity#/media/File:LowSensitivity\\_HighSpecificity\\_1400x1050.png](https://en.wikipedia.org/wiki/Sensitivity_and_specificity#/media/File:LowSensitivity_HighSpecificity_1400x1050.png)

**47.** <https://www.tensorflow.org/lite/images/convert/workflow.svg>

## **48. KFLASH**

## **49. MAIXPY IDE**

## **50. PUTTY**

**51.** [https://es.mathworks.com/help/examples/nnet/win64/SortClassesInAFixedOrderExample\\_01.png](https://es.mathworks.com/help/examples/nnet/win64/SortClassesInAFixedOrderExample_01.png)

**52.** [https://miro.medium.com/max/1400/1\\*uQDpo9iISx00ucl3gftLVA.png](https://miro.medium.com/max/1400/1*uQDpo9iISx00ucl3gftLVA.png)

**53,54,55,56.**

**53.** Fotos de la placa y componentes

**57.** Resultados del modelo 1

**58.** Fabricación Propia

**59,60.**

**59.** Resultados del modelo 1

**61.** [https://alexisbcook.github.io/assets/class\\_activation\\_mapping.png](https://alexisbcook.github.io/assets/class_activation_mapping.png)

**62,63,64,65.**

**62.** Imgenes del modelo 2

**66.** [https://blobscdn.gitbook.com/assets%2F-LGHUhl6VYqrZm4Re77O%2F-LGHYwvncirBGkeyLS8z%2F-LGH\\_1zvfJXzh6IBv8dZ%2FScreen%20Shot%202018-06-26%20at%204.57.26%20PM.png?alt=media&token=568a9ec5-1a55-4e63-8a9e-931e9ea21b1b](https://blobscdn.gitbook.com/assets%2F-LGHUhl6VYqrZm4Re77O%2F-LGHYwvncirBGkeyLS8z%2F-LGH_1zvfJXzh6IBv8dZ%2FScreen%20Shot%202018-06-26%20at%204.57.26%20PM.png?alt=media&token=568a9ec5-1a55-4e63-8a9e-931e9ea21b1b)

**67,68,69.**

**67.** Imagenes modelo final

**70-75.**

**70.** Imagenes de resultados de la red

**76.** TABLA PROPIA

**77.** [https://vignette.wikia.nocookie.net/reinoanimalia/images/b/b5/Le%C3%B3n\\_wiki2.png/revision/latest?cb=20130303082204&path-prefix=es](https://vignette.wikia.nocookie.net/reinoanimalia/images/b/b5/Le%C3%B3n_wiki2.png/revision/latest?cb=20130303082204&path-prefix=es)

**78.1**

**78.** [https://static.nationalgeographic.es/files/styles/image\\_3200/public/2928.600x450.jpg?w=1900&h=1425](https://static.nationalgeographic.es/files/styles/image_3200/public/2928.600x450.jpg?w=1900&h=1425)

**78.2**

**78.** [https://upload.wikimedia.org/wikipedia/commons/thumb/0/07/Edificio\\_Duque\\_de\\_Lerma\\_Valladolid.jpg/1200px-Edificio\\_Duque\\_de\\_Lerma\\_Valladolid.jpg](https://upload.wikimedia.org/wikipedia/commons/thumb/0/07/Edificio_Duque_de_Lerma_Valladolid.jpg/1200px-Edificio_Duque_de_Lerma_Valladolid.jpg)

**78.3**

- 78.** <https://concepto.de/wp-content/uploads/2018/08/persona-e1533759204552.jpg>

**78.4**

- 78.** [https://www.quecochemecompro.com/wor/cover\\_images/1175/450x380\\_1175.jpg?timestamp=1573029820](https://www.quecochemecompro.com/wor/cover_images/1175/450x380_1175.jpg?timestamp=1573029820)

**79.** Realización propia

**80.** Bash de Ubuntu WIN10

**81.** Bash Ubuntu transformación modelo

**82.** KFLASH Para firmware

**83.** Carga

**84.** Cargado correctamente

**85.** Carga modelo

**86, 87.**

**86.** Código

**88.** Imagen de montaje de LEDS

**89, 90, 91, 92, 93.**

**89.** Código

**94.** Resultado final

**95.** Putty inicial

**96,97,98**

**96.** uPyLoader capturas.

---

# Anexo I

## Configuración de nuestro equipo

---

Dada que la configuración de nuestro equipo tanto en software como en hardware es un factor importante para nuestro proyecto vamos a indicar estos componentes que incorpora nuestro ordenador portátil para futuras implementaciones con un equipo de parecidas características.

El ordenador usado para la realización del trabajo ha sido el [ASUS Rog Strix G731GU-H7154T](#), el cual se trata de un ordenador portátil gaming que cuenta con muy buenos componentes para realizar nuestro trabajo, destacando entre ellos:

- **16 GB de RAM DDR4 2666 MHz.** Los 16 gb de ram nos ayudaran a cargar las imágenes en nuestro script de entrenamiento y que se hagan las épocas de forma más rápida.
- **Procesador Intel i7-9750H (6 núcleos, 64 bits).** Contar con un procesador de última generación nos va a ayudar a realizar multitarea y a conseguir mejor rendimiento del equipo en general, consiguiendo hacer los diferentes pasos del trabajo sin tener muchos tiempos de espera.
- **NVIDIA® GeForce® GTX1660Ti 6GB GDDR6 VRAM.** La gráfica es uno de los componentes que más hace que varia el tiempo de entrenamiento. Esta gráfica de última generación nos ha permitido entrenar todas las imágenes en un tiempo aceptable, permitiendo hacer multitud de pruebas antes de llegar al modelo final.
- **Disco duro 512GB SSD M.2 PCIE NVME.** El contar con un disco duro SSD con tecnología M.2 nos permite contar con tiempos de transferencia muy bajos, ayudando con las tareas de carga y descarga de archivos desde este.
- **Refrigeración Gaming.** Al contar con un diseño Gaming conseguimos mantener unas temperaturas ideales para trabajar a máximo rendimiento durante el entrenamiento, evitando ralentizaciones y fallos por exceso de calor.

En cuestión de software necesitamos principalmente estos diferentes programas:

- **Anaconda (Jupyter Notebook).** Para entrenar nuestra red hemos utilizado la aplicación Anaconda, la cual nos incluye unas librerías perfectas para trabajar con tensorflow y las diferentes librerías de Python. En nuestro caso hemos usado la versión 2.1 de tensorflow, editando el código desde la aplicación Jupyter. [Descarga aquí](#)

- **Ubuntu (Para Windows).** Como hemos comentado en la sección de implementación hemos tenido que instalar la ventana de Ubuntu para usar el software necesario para hacer algunos pasos de la incorporación del modelo en la placa. [Descarga aquí](#)
- **Elementos del Github y dataset.** Para evitar poner todos los elementos necesarios en el proyecto los hemos recopilado todos en nuestro [repositorio del TFG](#).
- **Kflash GUI.** Este programa nos ayudará a cargar los archivos al interior de la placa. [Descarga aquí](#)
- **MaixPy IDE (Opcional).** Para editar el texto de micropython podemos usar el editor oficial, el cual nos permite ejecutar el código de manera fácil y sencilla. [Descarga aquí](#)
- **Firmware para la placa.** Por defecto viene incluido en la placa un firmware bastante antiguo, siendo necesario descargar las últimas versiones para que funcione todo correctamente. [Descarga aquí](#)
- **Putty.** Si queremos trabajar desde el interior de la placa necesitamos este programa para entrar a la ventana de comandos interna del dispositivo. [Descarga aquí](#)
- **uPyLoader.** Para modificar los archivos que se encuentran en el interior de la placa y hacer que funcione de forma automática debemos usar este programa. [Descarga aquí](#)

---

## Anexo II

# Pasos para la incorporación del modelo

---

Durante la memoria hemos explicado de forma más específica como se realizan los diferentes pasos para incorporar un modelo personalizado sobre nuestro dispositivo portátil.

Para que quede más claro y para que sirva de guía rápida vamos a numerar los diferentes pasos clave para pasar el modelo de forma correcta.

Empezaremos la guía una vez entrenado el modelo correctamente que cuente con las características necesarias para conseguir que funcione en nuestro dispositivo, es decir, que cuente con tantas neuronas de salida como clases a clasificar y que se use la función de activación *softmax*, junto a todo lo que ello conlleva.

Partiendo de este concepto y de los programas expuestos en el [Anexo I](#) podemos numerar los siguientes pasos:

1. **Transformación del modelo** guardado en formato “.h5” (keras) a “.tflite” (TensorFlow Lite). Este paso es necesario para transformar el modelo al formato de la placa.
2. **Reducción del modelo** al formato “.kmodel”, el cuál es el usado en el interior de la placa y el que es capaz de leer usando las diferentes librerías presentes en el firmware de la placa. Este paso se realiza usando el programa Ubuntu para Windows junto a los ficheros presentes en el repositorio.
3. **Carga del firmware dentro de la placa.** Usando el programa para cargar los diferentes ficheros dentro de la placa debemos cargar el firmware descargado.
4. **Carga del modelo.** De la misma forma que hemos cargado el firmware en la placa debemos cargar la versión “.kmodel” de nuestro modelo personalizado. Es importante apuntar la dirección de memoria en la cual hemos guardado el modelo.
5. **Creación de un script de micropython.** Necesitamos un pequeño script el cual se encargue de gestionar el modelo previamente cargado, siendo capaz de leer las imágenes del sensor y clasificarlas según el resultado del modelo.
6. **(Opcional) Automatización del modelo.** Para conseguir que el modelo se ejecute una vez enchufada la placa debemos modificar el fichero main en el interior de la placa, sustituyéndolo por nuestro script personalizado.