Clasificación y reconocimiento de imágenes con redes neuronales para entornos industriales.

Autor
Andrés Murgui Lozano
(anmurlo@alumni.upv.es)

Director: José Enrique Simó Ten
(jsimo@disca.upv.es)

TRABAJO FIN DE MASTER
MASTER DE AUTOMÁTICA E INFORMÁTICA INDUSTRIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
Valencia, 14/09/2018

Para mi futuro yo
cuando recuerde todo el trabajo invertido
y pueda reírse.

AGRADECIMIENTOS

Gracias a mi tutor José por estar ahí con las correcciones permitiendo que se terminase a tiempo.
A mi familia porque me han permitido intentarlo.
Y a Leena que me ha soportado en los días más difíciles de este proyecto con una sonrisa.

TABLA DE CONTENIDO

1. INTRODUCCION....10
2. JUSTIFICACION....11
3. OBJETIVOS....12
   3.1 Objetivo general....12
   3.2 Objetivos específicos....12
      3.2.1 Programar un sistema de generación de imágenes para entrenar....12
      3.2.2 Entorno de trabajo....12
      3.2.3 Entrenamiento....12
      3.2.4 Análisis rendimiento del sistema empotrado....12
4. ESTADO DEL ARTE....14
   4.1 Deep learning....14
      4.1.1 Hardware....14
         4.1.1.1 CPU....14
         4.1.1.2 RAM....14
         4.1.1.3 GPU....15
         4.1.1.4 Otros....15
      4.1.2 Software....16
         4.1.2.1 SO....16
         4.1.2.2 Lenguaje de programación....16
         4.1.2.3 Librerías....16
      4.1.3 Sistema empotrado....17
         4.1.3.1 Software del sistema empotrado....17
5. Desarrollo del proyecto....18
   5.1 Entorno de trabajo....19
      5.1.1 Workstation....19
         5.1.1.1 Hardware....19
         5.1.1.2 Software....21
      5.1.2 Sistema empotrado....23
         5.1.2.1 Hardware....23
         5.1.2.2 Software....24
   5.2 Modulos software de ENtrenamiento....26
      5.2.1 Módulo de Generación de Datos....26

5.2.2 Módulos de implementación del modelo.....27
5.2.2.1 Capas.....28
5.2.2.2 Capas de activación.....29
5.2.3 Modulo software de entrenamiento.....30
5.2.4 Resultado.....32
5.3 modulo software de adquisicion de datos y evaluacion del modelo.....33
5.3.1 Inicialización.....33
5.3.2 Adquisición de imágenes.....33
5.3.3 Preprocesamiento.....34
5.3.4 Evaluación.....34
5.3.5 Muestra de Resultados.....34
6. RESULTADOS.....35
6.1 Workstation.....35
6.1.1 Alexnet.....35
6.1.2 VGGnet.....37
6.1.3 VGGnet ampliada.....39
6.1.4 VGGnet ampliada sin generador de datos.....41
6.2 Sistema empotrado.....43
6.2.1 AlexNet.....43
6.2.2 VGGnet simple.....45
6.2.3 VGGnet aumentada.....46
6.2.4 VGGnet aumentada sin generador.....48
7. ANALISIS Y DISCUSION DE RESULTADOS.....50
7.1 Alexnet.....50
7.1.1 Entrenamiento.....50
7.1.2 Despliegue.....50
7.2 VGGnet Simple.....51
7.2.1 Entrenamiento.....51
7.2.2 Despliegue.....51
7.3 VGGnet Aumentada.....51
7.3.1 Entrenamiento.....51
7.3.2 Despliegue.....51
7.4 VGGnet aumentada sin generador.....52
7.4.1 Entrenamiento.....52

7.4.2 Despliegue....52
8. CONCLUSIONES....53
9. RECOMENDACIONES Y TRABAJOS FUTUROS....54
10. Trabajos citados....55
11. ANEXOS....56
11.1 Anexo 1 GenXUXEs....56
11.2 AnEXO 3 trainvgg.py....59
11.3 Anexo4 Alexnet....62
11.4 anexo 5 VGGnet simple....64
11.5 anexo 6 vggnet ampliada....66
11.6 anexo 7 PlendTFM....68

LISTA DE TABLAS

Tabla 1 Comparación de los sistemas empotrados lideres actualmente.....17
Tabla 2 Resultados de evaluación modelo Alexnet.....37
Tabla 3 Resultado evaluación del modelo VGGnet simple.....39
Tabla 4 resultados Validación VGGnet Ampliada.....40
Tabla 5 resultado de validación del modelo VGGnet Ampliada sin generador.....41
Tabla 6 Tabla de distribución de resultados.....43
Tabla 7 Tabla de distribución de resultados.....45
Tabla 8 Matriz de resultados con 150 epochs.....46
Tabla 9Matriz de resultados con 60 epochs.....47
Tabla 10 Matriz de resultados para modelo entrenado sin generador de datos.....48

LISTA DE FIGURAS

Ilustración 1 Workstation valorada en 11000$.....16
Ilustración 2 Diagrama del proceso de desarrollo.....18
Ilustración 3 especificaciones técnicas de la CPU.....19
Ilustración 4 Versión instalada de CUDA.....19
Ilustración 5 Versión instalada de CuDNN.....19
Ilustración 6 GPU usada en la Workstation.....20
Ilustración 7 Especificaciones de la GPU instalada.....20
Ilustración 8 memoria RAM instalada.....20
Ilustración 9 Características de la Workstation.....21
Ilustración 10 Placa Raspberry Pi 3B.....23
Ilustración 11 Cámara usada para la adquisición de imágenes.....24
Ilustración 12 ejemplo de uso de la interfaz IDLE para programación.....25
Ilustración 13 distribución del directorio Xuxes.....26
Ilustración 14 resultado en el directorio de salida.....27
Ilustración 15 ejemplo de imágenes generadas de cada clase.....27
Ilustración 16 grafica de una función ReLu.....29
Ilustración 17 Función Softmax.....29
Ilustración 18 Matriz de resultados de evaluación del modelo.....32
Ilustración 19 Estructura interna Modelo Alexnet.....35
Ilustración 20 Datos de entrenamiento Alexnet.....37
Ilustración 21 Esquema de la estructura del modelo VGGnet Simple.....38
Ilustración 22 Resultado entrenamiento de una VGGnet simple, durante 150 epoch.....38
Ilustración 23 Estructura del modelo VGGnet Ampliada.....39
Ilustración 24 Resultado del entrenamiento para una VGGnet ampliada durante 150 epochs 40
Ilustración 25 Resultados del entrenamiento sin generador de datos del modelo VGGnet ampliada.....41
Ilustración 26 Extracción de la consola durante la evaluación del modelo AlexNet.....44
Ilustración 27 Ejemplos de pruebas realizadas.....44
Ilustración 28 Extracción de la consola durante la evaluación del modelo VGGnet simple 45
Ilustración 29 ejemplo muestra de clase NoValido.....46
Ilustración 30 Extracción de la consola durante la evaluación del modelo VGGnet aumentado.....47
Ilustración 31 Extracción de la consola del modelo VGGnet aumentado sin generador.....48

RESUMEN EXTENDIDO

Con el desarrollo de la industria 4.0 se ha logrado acceso a gran cantidad de información y en consecuencia se han desarrollado nuevas tecnologías y métodos para aprovechar al máximo esta información.
Una de estas tecnologías son las redes neuronales profundas, que aplicadas a sistemas de visión artificial suponen un gran avance en las técnicas de control de calidad y supervisión de la producción en entornos industriales.

En este Proyecto se han combinado los sistemas de visión artificial basados en redes neuronales profundas, con plataformas hardware de bajo coste. Se han utilizado sistemas empotrados y entrenamiento de las redes fuera de línea para abaratar el coste. El resultado ha mostrado la viabilidad en ciertos ámbitos industriales de baja velocidad pero ha demostrado también la importancia de realizar el entrenamiento con información extraída directamente de la línea a supervisar.

Palabras clave: Vision artificial, Redes Neuronales Profundas, Sistemas empotrados, Deep learning

1. INTRODUCCION

Debido a la incesante evolución en las tecnologías de visión artificial en la actualidad se está evolucionando de un procesamiento de imágenes basado en operaciones y cálculos diseñados por los técnicos u operarios para satisfacer necesidades concretas, a un sistema basado en el entrenamiento de profundas redes neuronales de distintos tipos capaces de analizar gran cantidad de imágenes a gran velocidad y con buenos resultados, todo esto a cambio de la necesidad de una gran potencia de computación, especialmente en la fase de entrenamiento de la red.

Llegado a este punto se pretende comprobar si esta tecnología se puede adaptar para hacerla viable para empresas con una cantidad de recursos más limitada y de ser así que resultados es capaz de proporcionar.

Para ello se pretende separar el proceso de entrenamiento y de evaluación en 2 máquinas diferentes:

La primera de entrenamiento será un ordenador de sobre mesa con relativamente alta capacidad de procesamiento.

La segunda será el sistema de adquisición de imágenes y de evaluación de la red, comprendida por un sistema empotrado (Raspberry pi), y una cámara compatible (Raspicam).

2. JUSTIFICACION

El proyecto se ha realizado para comprobar hasta donde, un montaje de bajo coste, puede llegar en el campo de la visión artificial.

Con este estudio se comprobará como de viable es implantar sistemas de visión artificial basados en redes neuronales profundas, en pequeñas empresas de producción con limitado presupuesto, acercando tecnología, de normal altamente costosa, a un precio asumible para estas empresas.

Para comprobar la viabilidad se ha partido del sistema empotrado Raspberry pi 3b, basado en Linux y con una amplia comunidad que respalda el desarrollo de aplicaciones en este sistema supone una gran base sobre la que construir un proyecto de estas características.

3. OBJETIVOS

3.1 OBJETIVO GENERAL

Comprobar la capacidad de un sistema empotrado de bajo coste (Raspberry pi) para implementar y utilizar un sistema de visión artificial basado en redes neuronales para detectar objetos en una línea de producción.

Adquirir la mayor cantidad de conocimientos posibles en este campo de la tecnología que se encuentra en auge gracias a la implantación de la industria 4.0 y otros sistemas de análisis de toda la información generada por las actividades industriales

3.2 OBJETIVOS ESPECÍFICOS

3.2.1 Programar un sistema de generación de imágenes para entrenar

Dado que no se tiene acceso a un sistema industrial objetivo, la información necesaria para entrenar la red se deberá obtener de manera artificial mediante un programa que mezcla fondos con objetos a detectar modificados, escalados y rotados de forma aleatoria

3.2.2 Entorno de trabajo

Tanto la Workstation en la que se hace el entrenamiento, como el sistema empotrado que lo va a aplicar, necesitan tanto software como librerías específicas varias de las cuales han de ser compiladas específicamente para cada máquina además de adaptarla a las necesidades y capacidades de cada sistema. Estos deberían ser lo más fácilmente replicables posible.

3.2.3 Entrenamiento

El objetivo es conseguir un sistema de entrenamiento que permita flexibilidad en los parámetros y métodos para así poder conseguir el mayor número de modelos posible además de obtener telemetría suficiente como para sacar información que sea de utilidad para analizar la situación del proyecto.

3.2.4 Análisis rendimiento del sistema empotrado

Debido a que los sistemas empotrados no tienen la misma potencia que un ordenador de sobremesa es importante conocer los límites de funcionamiento y si es viable desplegar una red neuronal en una Raspberry que sea capaz de dar un resultado satisfactorio

3.2.5

4. ESTADO DEL ARTE

En el campo del análisis de imágenes mediante técnicas de Deep learning hay que separar los dos factores limitantes, y por lo tanto las fronteras de la investigación tecnológica, en la actualidad: hardware y software

4.1 DEEP LEARNING

Deep learning es un conjunto de técnicas que buscan procesar información de cualquier tipo a partir de un entrenamiento del algoritmo. Este entrenamiento según lo que se busque hacer con él puede ser supervisado (clasificación) o no supervisado (clustering). En el caso de este proyecto se va a utilizar técnicas de Deep learning propias de imágenes, en concreto se utilizara un entrenamiento supervisado ya que el objetivo es identificar y clasificar las imágenes según su contenido.

4.1.1 Hardware.

Dentro del hardware se juntan muchos factores que en resumen determinan el tiempo necesario para entrenar un modelo determinado. La efectividad final de un modelo no depende del método o el hardware usado para entrenarlo, solo variara el tiempo que se habrá necesitado para llegar a ese mismo modelo.

4.1.1.1 CPU.

La CPU del ordenador a usar es relevante en caso de que se vaya a usar como única fuente de entrenamiento. Sin embargo la relación entre el número de núcleos junto con la velocidad del reloj no es proporcional a la velocidad que consigue entrenando [1] [CITATION eficiencia_CPU \l 3082]. Sin embargo no es conveniente usar una CPU demasiado limitada ya que aunque no se encargue del entrenamiento en sí, es necesaria para el preprocesamiento y carga de la información en la memoria RAM y VRAM.

4.1.1.2 RAM.

Al igual que en el caso anterior la cantidad de memoria RAM es relevante en el caso de realizar entrenamiento mediante CPU aunque sin por ello disponer de un mínimo que permita un rápido flujo de datos al proceso.

4.1.1.3 GPU.

La mayor parte del entrenamiento de redes neuronales en la actualidad se hace con tarjetas Gráficas o GPU que de normal se usan para el cálculo de gráficos en videojuegos o diseño gráfico. Estas tarjetas son piezas de hardware optimizadas en general para hacer un gran número de operaciones con la información almacenada en la VRAM, una sección de memoria localizada físicamente en la tarjeta cerca del núcleo de computación. Dentro de las características de una tarjeta gráfica que afectan a la velocidad de entrenamiento cuentan tanto la cantidad de VRAM disponible como la velocidad de reloj de la misma. En particular la cantidad de memoria disponible es el factor más importante ya que la limitación de las GPU es su lenta velocidad de transferencia de información por lo que la capacidad de tener tanto el modelo a entrenar como el conjunto de datos cargados por completo en la VRAM permite una gran velocidad mientras que si tiene que ir cargando datos nuevos en la misma, la eficiencia será mucho más limitada.

4.1.1.3.1 CUDA.

Aunque técnicamente CUDA es un componente software, está altamente ligada al funcionamiento de la GPU. CUDA es una librería optimizada para la computación en paralelo desarrollada por el fabricante NVIDIA para sus GPUs. Esta librería, además de estar soportada por las principales herramientas de computación de Deep learning, supone un incremento en el rendimiento de las GPUs de hasta el 50%, siendo esto lo que hace realmente rentable el uso de GPUs como medio de computación.

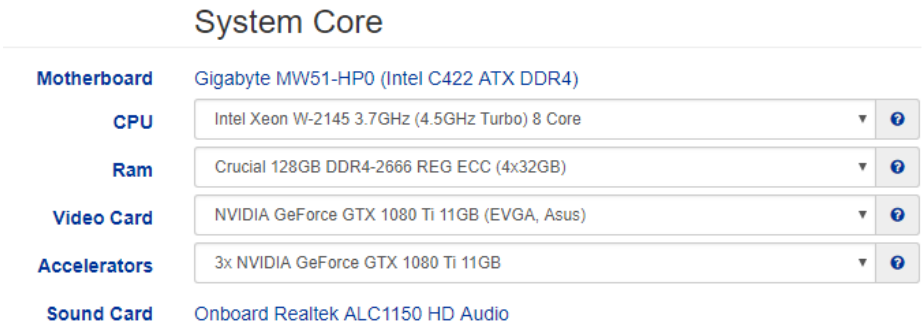
Esto limita al uso de GPUs a las fabricadas por NVIDIA, ya que el siguiente fabricante de GPUs del mercado, aunque también tiene librerías similares optimizadas para sus propias gráficas, estas son raramente soportadas por otro software, haciendo que el esfuerzo necesario para ponerlas en marcha deje de hacerlas rentables.

4.1.1.4 Otros.

Otros aspectos del Hardware típico de un ordenador como la cantidad de memoria disponible en el disco duro o la conexión a internet no son relevantes para la velocidad de entrenamiento del modelo

En la actualidad existen en el mercado diferentes opciones de Workstation (Ilustración 1) con potencia suficiente para llevar a cabo la mayoría de las actividades relacionadas con redes neuronales en el campo de la visión artificial aunque recientemente se han puesto de moda el alquiler de hardware como servicio como Amazon AWS que permite el alquiler de espacio de computación de un servidor con acceso a más o menos cantidad de recursos en función del precio.

Ilustración 1 Workstation valorada en 11000$



4.1.2 Software

4.1.2.1 SO.

Debido a la flexibilidad que ofrece para usuarios avanzados y una amplia comunidad que mantiene el código, usar Ubuntu 16.04 (ilustración 2) es la opción as recomendada y que sigue vigente incluso para sistemas recientes. Esto se debe principalmente a que es un SO ya testeado y que no va a ser actualizado de manera que afecte al funcionamiento general, esto junto a la gran cantidad de software ampliamente testeado, permite a la comunidad partir de una base sólida para desarrollar sus proyectos sin tener que preocuparse de dificultades extra a solventar

4.1.2.2 Lenguaje de programación

Aunque existe la posibilidad de implementar la red neuronal en cualquier lenguaje de programación, Python destaca por encima del resto por la cantidad, calidad y facilidad de uso de las librerías existentes al alcance de cualquier usuario y es ampliamente utilizado tanto por la comunidad científica como de aficionado para este tipo de proyectos.

4.1.2.3 Librerías

Entre ellas cabe destacar TensorFlow, una librería que optimiza la operación con tensores (vectores multidimensionales) con métodos para diseñar, entrenar y monitorizar modelos de redes neuronales, además de implementar soporte para CUDA para mejorar la velocidad de cálculo.

Como complemento existen herramientas como NumPy que suministra una interfaz para el tratamiento de matrices o PIL que dispone de métodos de adquisición y comandos básicos de procesamiento de imágenes. En caso de que se quieran realizar operaciones avanzadas con imágenes existe la librería OpenCV, esta librería está disponible para los principales sistemas operativos y lenguajes de programación, pero en el ámbito del proyecto resulta interesante porque se puede compilar de forma personalizada para que utilice la librería CUDA y CuDNN (especialización de CUDA para Deep Neural Network)

4.1.3 Sistema empotrado

Para el proyecto se ha elegido la Raspberry pi modelo 3b aunque actualmente ya ha salido a la venta un aversión mejorada (modelo 3b+) con mayor capacidad de procesamiento. Además gracias al empujón generado por la Raspberry otros competidores están desarrollando sistemas similares con incluso más potencia como por ejemplo Hikey 960 o ODROID-C2 (tabla 1).

Tabla 1 Comparación de los sistemas empotrados lideres actualmente

	Raspberry pi 3b	Raspberry pi 3b+	ODroid-C2	HIKEY 960
CPU	4x1.2GHz 64bits	4x1.4GHz 64 bits	4x2GHz 64 bits	4xA73+4xA53
GPU	Videocore-IV	Videocore-IV	ARM Mali-450	ARM Mali G71 MP8
RAM	1GB DDR2	1GB DDR2	2GB DDR3	3GB DDR4
Precio	29.47	29.47	40	225

4.1.3.1 Software del sistema empotrado

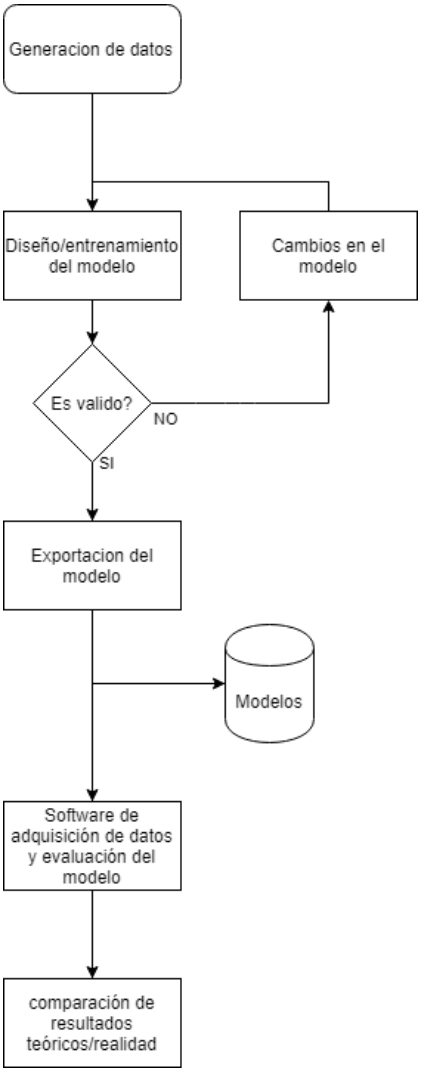
El S.O. utilizado en estos casos para la Raspberry pi es el Raspbian Stretch, una variante de Debian 9, optimizada y adaptada para comunicarse de forma más sencilla con las características del hardware.

5. DESARROLLO DEL PROYECTO

En este proyecto se pretende desarrollar un conjunto de software compatible con un sistema empotrado que acerque los sistemas de visión artificial a la pequeña industria. Para ello se pretende separar el costoso entrenamiento del software de identificación con el objetivo de evitar replicar el hardware necesario para entrenar los modelos en varios dispositivos, comprobando si un sistema empotrado de bajo coste es capaz de efectuar la tarea de evaluación por sí mismo.

Para ello se ha se ha establecido el proceso a seguir (Ilustración 2) y los módulos software que se necesitan desarrollar para cumplir estos objetivos.

Ilustración 2 Diagrama del proceso de desarrollo



El proyecto se ha dividido en 3 fases, construcción del entorno de trabajo (tanto en la Workstation como en la Raspberry pi), entrenamiento y validación. A continuación se explicara cómo se ha procedido en cada una de estas fases y se justificara por que se ha procedido de esa manera y no de otra.

5.1 ENTORNO DE TRABAJO

Antes de poder proceder a desarrollar los módulos software específicos para la realización del trabajo, es necesario definir los entornos en los que los mismos se van a desarrollar y desplegar para evitar conflictos y evitar problemas externos que entorpezcan el trabajo de programación y diseño.

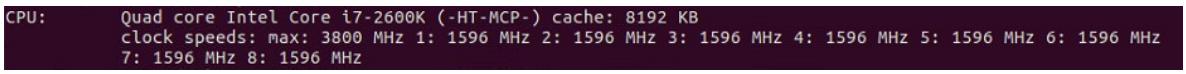
5.1.1 Workstation

5.1.1.1 Hardware

A continuación se describe el hardware usado en la Workstation para llevar a cabo el entrenamiento

5.1.1.1.1 CPU. Se ha utilizado un chip I7 de Intel modelo 2600k (Ilustración 3), esta CPU permitirá el preprocesado de las imágenes sin problema y ya que no se utilizara para el entrenamiento no supondrá un cuello de botella para el entrenamiento.

Ilustración 3 especificaciones técnicas de la CPU



5.1.1.1.2 GPU. Para la realización de este proyecto se ha buscado poder utilizar computación acelerada por hardware y para ello se ha escogido una tarjeta gráfica compatible con las librerías CUDA v9.2 (ilustración 4) y CuDNN v7.1.4 (ilustración 5).

Ilustración 4 Versión instalada de CUDA

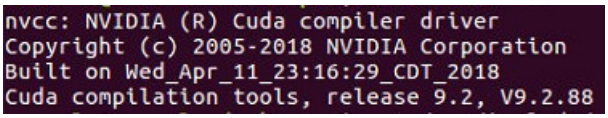


Ilustración 5 Versión instalada de CuDNN

El modelo elegido finalmente fue Geforce GTX 1050 (Ilustración 6) de la marca NVIDIA compatible con las librerías instaladas. Este modelo dispone de 2GB de VRAM y 1.4GHz (ilustración 7)

.

Ilustración 6 GPU usada en la Workstation

Ilustración 7 Especificaciones de la GPU instalada

```
Sat Sep 15 19:29:50 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.51                 Driver Version: 396.51                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1050    Off  | 00000000:01:00.0  On |                  N/A |
| 40%   35C    P8    N/A /  75W |    295MiB /  1999MiB |      2%      Default |
+-------------------------------+----------------------+----------------------+
```

5.1.1.1.3 RAM. La cantidad de RAM instalada son 4 sticks de 4GB DDR3 cada una, esto es más que suficiente para almacenar todas las imágenes usadas en el entrenamiento a la vez mientras se prepocesan. Además Debian como característica predeterminada crea un espacio de Swap de 15 GB (Ilustración 8) que en caso de necesidad se puede activar, a coste de menor velocidad y aumentar el tráfico en el disco duro.

Ilustración 8 memoria RAM instalada

```
               total        used        free      shared  buff/cache   available
Memoria:         15G        1,4G         13G         26M        1,0G         13G
Swap:            15G          0B         15G
```

En general la Workstation (Ilustración 9), aunque un poco limitada en la memoria disponible de la tarjeta gráfica, debería de ser capaz de entrenar redes neuronales no excesivamente complejas en una cantidad de tiempo relativamente pequeña.

Ilustración 9 Características de la Workstation

5.1.1.2 Software

La mayoría del software necesario para empezar a trabajar, al ser open source o de rápida evolución suele requerir que el usuario compile la librería en su máquina para así evitar errores de compatibilidad que podría tener un archivo ya compilado en otra máquina. Esto a cambio implica que para empezar a trabajar en los modelos y entrenamiento hay que, primero, conseguir compilar el software, lo que en determinados casos no es una actividad trivial y repleta de pasos que pueden hacer perder una tarde de compilación. A continuación se detallara el software principal usado en el desarrollo del proyecto:

5.1.1.2.1 S.O. Como sistema operativo se ha elegido la distribución de Linux: UBUNTU, en su versión 16.04. Aunque esta versión no es la más reciente, la actual es v18.04, si es la versión con una comunidad más amplia que permite solucionar los errores típicos que conlleva compilar las librerías necesarias.

5.1.1.2.2 CUDA. Una vez instalados los últimos drivers de NVIDIA para la GPU se procedió a compilar los binarios de la librería CUDA, para ello es necesario tener los drivers instalados y que la tarjeta sea compatible, una vez comprobado la instalación es sencilla y solo necesita que se establezcan las rutas de los binarios para evitar problemas de dependencias.

5.1.1.2.3 CuDNN. El siguiente paso es compilar el paquete CuDNN el cual requiere además que la tarjeta a usar tenga una potencia mínima según una escala de capacidad de computación propia de NVIDIA. El mínimo es un factor de puntuación de 3, la tarjeta actual tiene un factor de 6 con lo que el paquete s puede compilar e instalar sin más contratiempos.

5.1.1.2.4 TensorFLow. Esta es una de las librerías que es importante configurar bien la compilación ya que si no se activa la opción de utilizar CUDA, no aprovecharemos la potencia de las gráficas a la hora de entrenar el modelo. Esta librería es la que se encarga de crear, calcular y optimizar el modelo que después identificara las imágenes así que es importante que esta instalación transcurra sin errores y esté correctamente configurada

5.1.1.2.5 OpenCV. Se instalara la versión 3.4.1 que es la más reciente a fecha de realización de este proyecto. Debido a la gran cantidad de contenido que esta librería aporta, sobre todo si se incluyen las contribuciones de 3os, esta librería puede tardar fácilmente entre 3 y 4 horas en compilar. Además hay que asegurarse, al igual que con TensorFlow, que se compila con las dependencias correctas de CUDA y CuDNN y teniendo como objetivo la librería para Python.

5.1.1.2.6 Keras. Se instalara la versión 2.2 que es la más reciente disponible a fecha de realización de este trabajo. Keras es una librería de abstracción para TensorFLow, agrupa y simplifica la funcionalidad de TensorFlow permitiendo un desarrollo más ágil de modelos y su entrenamiento. A efectos prácticos la API que se usara durante el proyecto será la de Keras y en ningún momento se usara directamente TensorFlow, aunque sigue siendo necesario tenerlo instalado de forma correcta.

De manera complementaria a Keras se instalan dos paquetes para Python necesarios para explotar determinadas características de Keras.

El paquete H5PY nos permite generar un archivo que contiene el modelo, los pesos de las neuronas y la configuración de entradas salidas, con capacidad de trasladarlo a un sistema completamente diferente, y, mientras este nuevo sistema tenga Keras instalado este modelo debería de poder abrirse y utilizarse libremente.

Pydot. Este paquete nos permitirá generar gráficos con la información extraída del proceso de entrenamiento y validación.

5.1.1.2.7 IDE. Para el IDE a utilizar depende de los gustos del programador principalmente. En el proyecto se ha utilizado Pycharm por su sencilla integración con github y las herramientas de debug que dispone.

Se ha optado por usar Python como lenguaje de programación al igual que una versión desactualizada del sistema operativo, porque es el entorno donde más recursos existen y donde hay una más amplia comunidad que puede ayudar en caso de encontrar algún problema.

5.1.2 Sistema empotrado.

A diferencia de la Workstation el sistema empotrado ha de ser una pieza de hardware de bajo coste y que resulte sencillo replicar las veces necesarias para acomodar la funcionalidad a todos los sitios que sea necesarios

5.1.2.1 Hardware

5.1.2.1.1 Placa principal. Se ha elegido la placa Raspberry pi 3B Ilustración debido a su equilibrio entre coste, y prestaciones. Además es muy sencillo conseguir una incluso en tiendas de electrónica y de telefonía. La comunidad alrededor de esta plataforma es muy activa y abarca una gran cantidad de campos diferentes con lo que es fácil que cualquier problema que pueda surgir ya le haya pasado a alguien antes y además alguien más lo haya solucionado.

Obviamente el hecho de ser un sistema empotrado nos presenta unas limitaciones claras como por ejemplo no disponer de una tarjeta gráfica con la que hacer los cálculos, sin embargo debido a que solo se tienen que hacer evaluaciones unitarias de imágenes no es tan crítico el uso de una GPU y puede ser suficiente con la potencia proporcionada por el procesador.

Ilustración 10 Placa Raspberry Pi 3B

5.1.2.1.2 Cámara. Como método de adquisición de las imágenes se utiliza una cámara que se conecta directamente al puerto CSI de la Raspberry. Esta cámara con una resolución de 5Mpixeles y con capacidad de capturar una imagen de hasta 1080p a 30 FPS al estar conectada directamente al bus de datos y no a través de un USB hará la adquisición de imágenes mucho más rápida dejando más tiempo para la evaluación del modelo.

Ilustración 11 Cámara usada para la adquisición de imágenes

5.1.2.2 Software

Debido a que los modelos han de funcionar tanto en la Workstation como en el sistema empotrado las versiones del software han de ser como mínimo compatibles e idealmente, iguales. Esto no siempre es posible ya que al trabajar con recursos limitados pueden haber funcionalidades que no se puedan implementar, a continuación se mostraran principalmente las diferencias con el software instalado en la Workstation.

5.1.2.2.1 S.O. en este caso el sistema operativo cambia por completo, aunque tanto Ubuntu como Raspbian ambos son modificaciones de la distro original Debian por lo que a pesar de diferencias eventuales la mayoría del software es compatible sin mayor complicación. En este caso se ha instalado Raspbian Stretch desktop, siendo la versión más reciente. Sea optado por la versión con escritorio por la comodidad de colocar una pantalla y pre visualizar los resultados.

5.1.2.2.2 CUDA y CuDNN. En este caso al no tener tarjeta gráfica y siendo que el chip grafico no pertenece a NVIDIA, no se han podido instalar ni CUDA ni CuDNN y por lo tanto en el resto del software no se han usado sus dependencias.

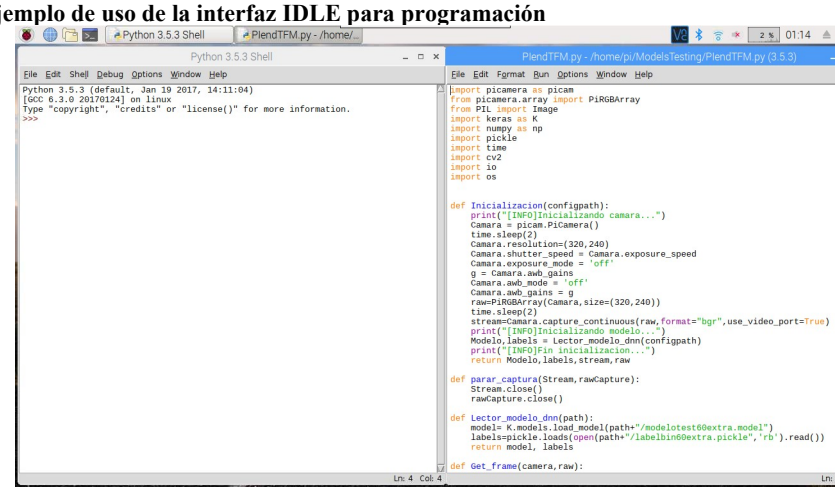
5.1.2.2.3 TensorFlow. Debido a que no era necesario usar aceleración por hardware se ha optado por instalar una versión precompilada mediante la herramienta PiP lo que simplifica enormemente el proceso y permite el cambio de versiones de forma sencilla y rápida.

5.1.2.2.4 OpenCV. No hay disponibles una gran variedad de versiones de OpenCV precompiladas listas para poder instalar y esto es debido a lo personalizable que es la compilación. Debido a esto se ha tenido que compilar en la Raspberry la librería desde 0 a partir de las fuentes, lo cual fuerza a implementar un swap space para que el proceso no se quede sin memoria y pueda terminar sin problemas.

5.1.2.2.5 Keras. Tanto Keras como los paquetes extra se instalan exactamente igual que en el caso de la Workstation mediante PiP

5.1.2.2.6 IDE. En el caso del IDE para la Raspberry, la misma distribución incorpora 3 IDEs compatibles con Python, del cual se ha usado IDLE (Ilustración12), una simple consola de línea de comandos de Python con un editor de texto incluido, el cual dispone de un acceso rápido a ejecutar programa.

Ilustración 12 ejemplo de uso de la interfaz IDLE para programación



5.2 MODULOS SOFTWARE DE ENTRENAMIENTO

Una vez establecido el entorno de trabajo se va a empezar con el desarrollo de programas orientados al entrenamiento de los modelos. Esta parte se desarrolla exclusivamente en la Workstation.

5.2.1 Módulo de Generación de Datos

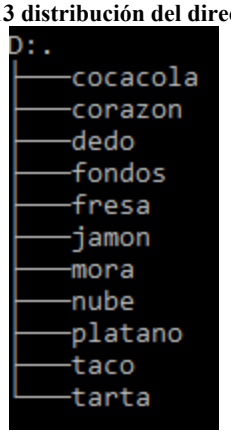
Debido a la dificultad de obtención de datos reales, se ha creado un programa (Anexo1) que se encarga, a partir de una serie de fondos y elementos a identificar, generar imágenes con los objetivos escogidos aleatoriamente de entre todos los disponibles de su clase, y colocados y escalados de forma también aleatoria dentro de la zona del fondo.

El programa tiene la capacidad también de generar un archivo de texto para cada imagen con información útil para el entrenamiento, como la etiqueta correcta y la bounding box correspondiente al objetivo.

EL programa funciona gracias a la organización de los recursos dentro del directorio de trabajo (Ilustracion13).

Esto permite aumentar fácilmente la variabilidad de las imágenes generadas solo con añadir las imágenes correspondientes a cada carpeta. Como mínimo debe haber un elemento pero no hay máximo y no hay que cambiar el código para que los cambios sean efectivos.

Ilustración 13 distribución del directorio Xuxes



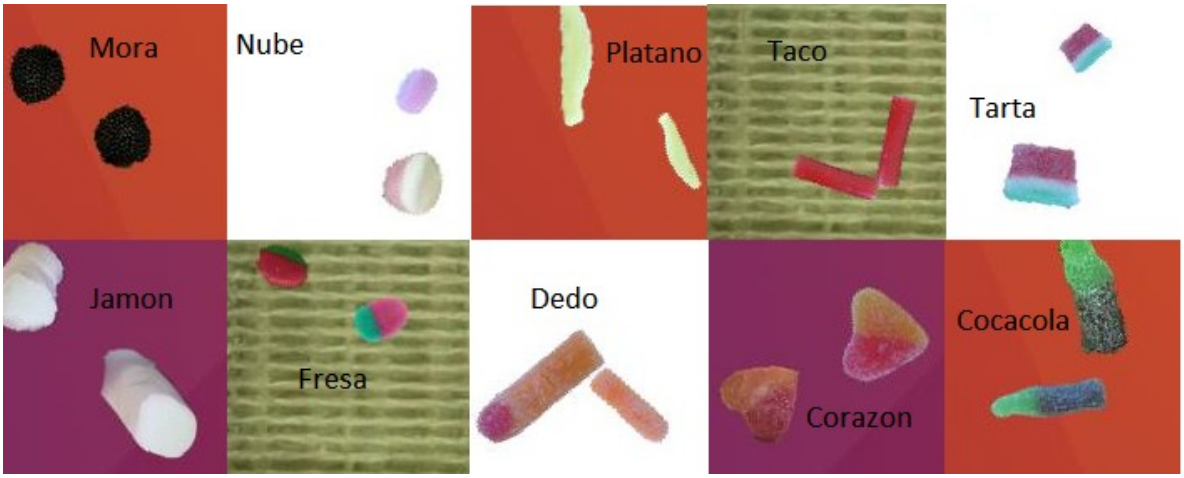
Las entradas necesarias para el funcionamiento del programa son:

- Nombre de la carpeta de la chuchería número 1 que añadir
- Nombre de la carpeta de la chuchería número 2 que añadir, o vacío en caso de que se desee incluir solo una chuchería en la imagen
- Numero de imágenes diferentes que se pretende generar
- Directorio donde se quieren guardar los resultados
- Si se quiere generar los archivos con información o no.

Con esta herramienta se pueden generar una gran cantidad de Datos en muy poco tiempo, como por ejemplo las 60500 imágenes (Ilustración 14), de las 11 clases diferentes (Ilustración 15), generadas para el entrenamiento de los modelos

Ilustración 14 resultado en el directorio de salida

Ilustración 15 ejemplo de imágenes generadas de cada clase



5.2.2 Módulos de implementación del modelo

Los modelos están programados, cada uno, en un archivo independiente (Anexos 4, 5 y 6) para facilitar su modificación de forma independiente al código de entrenamiento.

Estos modelos, aunque se basan en modelos ya estudiados y con un funcionamiento comprobado con datasets de competición [CITATION Fei17 \l 3082], se han hecho repetidas modificaciones hasta encontrar las que mejor resultado de entrenamiento ofrecían, aunque eso no implica que sea el mejor resultado posible.

Cada modelo está compuesto por 2 elementos principales: las capas y las funciones de activación, aunque estas últimas también se aplican como capas. Su función es interpretar los resultados de la capa anterior y normalizar los valores.

5.2.2.1 Capas

La forma de decidir el tipo y cantidad de capas que lleva un modelo depende mayoritariamente de prueba y error, más que de un método probado, aunque sí que hay unas pautas y una tendencia actual.

5.2.2.1.1 Convolucion2D.Se han utilizado conjuntos de capas de convolucion, las cuales analizan por cada pixel también los de alrededor por lo que ya no solo cuenta el valor del pixel sino también de que está rodeado el mismo para determinar el valor final de la neurona, esto permite una percepción espacial de la información.

5.2.2.1.2 Dropout. También se han añadido capas de dropout, las cuales de forma aleatoria durante el entrenamiento interrumpen la propagación de la información a la siguiente capa en esa neurona, esto ayuda a que ninguna neurona se sobreentrene y no sea útil a la hora de analizar algo que no pertenezca al set de entrenamiento.

5.2.2.1.3 BatchNormalization. Normaliza los valores de entrada restándole la media y dividendo entre la desviación estándar de los valores normalizados. Esto permite aumentar los pasos de aprendizaje y en general agiliza el entrenamiento

5.2.2.1.4 MaxPooling. Esta capa reduce las dimensiones de la matriz de entrada manteniendo las dimensiones de entrada. Según el tamaño de la ventana agrupa valores de los pixeles de entrada en una salida con el valor máximo de los pixeles agrupados

5.2.2.1.5 Flatten. Flatten transforma una entrada bidimensional como la que sale de una capa de convolucion o MaxPooling y la transforma en un vector de una sola dimensión. Esto es útil para para utilizar otro tipo de capas o como paso previo a terminar el modelo

5.2.2.1.6 Dense. También conocidas como “fully connected” son capas de neuronas en las que cada una de las neuronas está conectada a todas las salidas de la capa anterior. Son el diseño más clásico de red neuronal cada vez más en desuso.

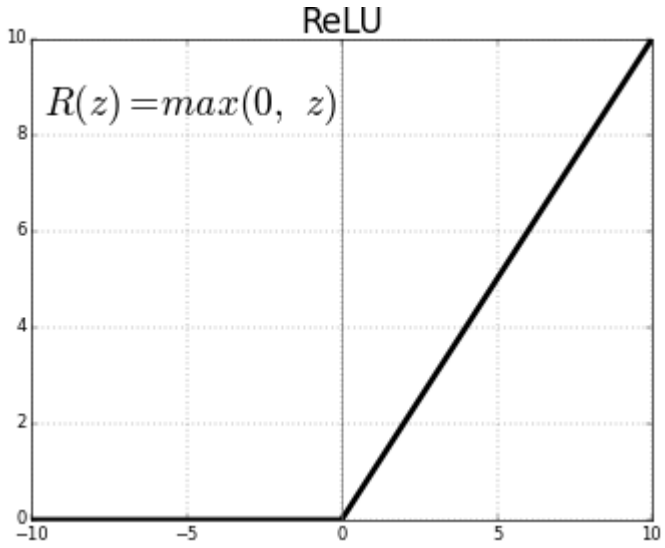
5.2.2.2 Capas de activación

Las capas de activación se colocan después de las capas de convolucion o Densas con el objetivo de normalizar la salida de estas capas, el tipo de capa determina la función que se utiliza para normalizar. Las más conocidas son:

5.2.2.2.1 ReLu.

Es la función de activación más usada en las principales redes neuronales La más común es la RELU (Ilustración 16) esta se basa en la función R(x)= (Max (0, x)). Esta función evoluciona rápidamente sin embargo no es adecuada para usarla en la última capa del modelo ya que te limita el rango de salida y en una aplicación de clasificación como la nuestra no es conveniente

Ilustración 16 grafica de una función ReLu



5.2.2.2.2 Softmax.

La función Softmax (Ilustración 17) se utiliza principalmente como capa de salida, ya que transforma la información de entrada a la capa en un vector de probabilidades, cuyo valor final suma 1, que permite decidir la clase en función de la confianza que el modelo devuelve.

Ilustración 17 Función Softmax

$$\sigma : \mathbb{R}^K \to [0,1]^K$$

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

5.2.3 Modulo software de entrenamiento

Una vez decidido el modelo que se va a usar y con acceso a las imágenes se puede proceder a entrenar el modelo.

El código de entrenamiento (Anexo 3) se encarga de leer las imágenes generadas anteriormente, adjudicarle etiquetas en función del directorio del que se han extraído y reescalar al tamaño de manipulación.

```
# mezcla el dataset para evitar aprender secuencias
imagePaths = sorted(list(paths.list_images(args["dataset"])))
random.seed(42)
random.shuffle(imagePaths)

# preprocesamiento de todas las imagenes
for imagePath in imagePaths:
          # lee la imagen y la modifica a 64X64 para entrar en la red diseñada
          image = cv2.imread(imagePath)
          image = cv2.resize(image, (64, 64))
          data.append(image)

          # a partir del path de la imagen podemos averiguar la etiqueta correspondiente
          # se añade a la lista de labels
          label = imagePath.split(os.path.sep)[-2]
          labels.append(label)
```

Estos datos son divididos en 3 grupos para las diferentes fases del entrenamiento. se genera el intérprete de las clases y se aplica a la lista de etiquetas de cada grupo para traducirla a un vector que el modelo pueda entender.

```
# DIvision de los datos en 3 grupos 60/20/20%
(trainX, tempX, trainY, tempY) = train_test_split(data,
          labels, test_size=0.40, random_state=42)
(testX, valX, testY, valY) = train_test_split(tempX,
          tempY, test_size=0.50, random_state=42)

#convertimos el total de posibles clases en un vector de 1y 0
#este vector será el intérprete entre el mundo y la red.
lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)
valY = lb.transform(valY)
```

A continuación se carga en memoria el modelo y se inicializan tanto el generador de datos, como los valores de configuración del entrenamiento

```
# Aquí se inicializa el ampliador de datos para el entrenamiento,
#especificando el tipo de modificaciones que se pueden aplicar.
aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
          height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
          horizontal_flip=True, fill_mode="nearest")

# Se carga en memoria el modelo seleccionado.
model = SmallVGGNet.build(width=64, height=64, depth=3,
          classes=len(lb.classes_))

# se inicializa el paso de entrenamiento, el número de epochs deseado
# y el tamaño de cada batch
INIT_LR = 0.01
EPOCHS = 100
BS = 32
```

De los parámetros más relevantes son el número de epochs que se van a efectuar, siendo una epoch una cantidad arbitraria de imágenes diferentes usadas para entrenamiento. De normal se considera una epoch cuando todas las imágenes de entrenamiento pasan por el modelo. El número de epochs determinara lo lejos que llegue el entrenamiento del modelo, siendo al principio conveniente fijar una cantidad mayor de la que sería predecible y a partir de los resultados del entrenamiento ir bajando el número de epochs hasta afinar el resultado.

Otro parámetro relevante es el paso de entrenamiento, que fija cuanto pueden llegar a cambiar los parámetros en cada iteración, este valor puede determinar si el modelo llega a una solución o simplemente no es capaz de encontrar unos valores adecuados que devuelvan los valores correctos al evaluar el modelo.

BS, o batch size, determina la cantidad de imágenes que se utilizara por cada paso de entrenamiento hasta terminar la epoch. Conviene que sea un valor mayor que el número mínimo de clases diferentes y no lo suficientemente grande como para que el tiempo de carga de las imágenes sea demasiado largo.

Una vez hecho esto se determina la función de optimización y la de error y el entrenamiento puede dar comienzo.

```
# SE inicializa el optimizador del modelo usado en el entrenamiento,
# asi como el modelo en si especificando la función de error.
print("[INFO] Entrenando red...")
opt = SGD(lr=INIT_LR, decay=INIT_LR / EPOCHS)
model.compile(loss="categorical_crossentropy", optimizer=opt,
          metrics=["accuracy"])
#se guarda el esquema del modelo entrenado
plot_model(model,to_file="/home/anmurlo/Resultados_modelos/esquema60extranogen.png")

# empieza el entrenamiento del modelo
```

```
H = model.fit(trainX, trainY, batch_size=BS,
        validation_data=(testX, testY), epochs=EPOCHS
```

5.2.4 Resultado

El mismo código que se dedica a entrenar el modelo, hace recopilación de datos y guarda, en una imagen, los valores tanto de error como de aciertos de ambos grupos (entrenamiento y test).

```
# se genera y guarda la gráfica de error y precisión
# a partir del historial generado durante el entrenamiento
N = np.arange(0, EPOCHS)
plt.style.use("ggplot")
plt.figure()
plt.plot(N, H.history["loss"], label="train_loss")
plt.plot(N, H.history["val_loss"], label="val_loss")
plt.plot(N, H.history["acc"], label="train_acc")
plt.plot(N, H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy (SmallVGGNet)")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.savefig(args["plot"])
```

Como último paso una vez ha terminado el entrenamiento pasa el último set de imágenes, el de validación, que aún no había sido usado y genera una matriz con datos (ilustración 18) de precisión para cada una de las clases que componen el modelo.

```
predictions = model.predict(valX, batch_size=32)
print(classification_report(valY.argmax(axis=1),
        predictions.argmax(axis=1), target_names=lb.classes_))
```

Ilustración 18 Matriz de resultados de evaluación del modelo

	precision	recall	f1-score	support
NoValido	0.93	0.71	0.81	1109
cocacola	0.97	1.00	0.98	1097
corazon	0.97	1.00	0.98	1079
dedo	0.97	0.99	0.98	1153
fresa	0.97	1.00	0.98	1100
jamon	0.96	1.00	0.98	1079
mora	0.96	1.00	0.98	1145
nube	0.98	0.99	0.99	1061
platano	0.98	1.00	0.99	1090
taco	0.99	0.98	0.98	1123
tarta	0.97	1.00	0.98	1064
avg / total	0.97	0.97	0.97	12100

5.3 MODULO SOFTWARE DE ADQUISICION DE DATOS Y EVALUACION DEL MODELO

Este programa (Anexo 7) se encarga de obtener una vía directa a la información capturada por la cámara y hacer pasar la información obtenida a través del modelo, que devolverá la clase a la que corresponde esa imagen. Para ello se sigue el siguiente proceso

5.3.1 Inicialización.

Este es el proceso más largo ya que se tiene que inicializar la cámara, dejar tiempo para que el sensor se prepare para capturar, y fijar los parámetros para que todas las imágenes se capturen de igual manera a lo largo de la ejecución. Además hay que cargar en memoria el modelo y los pesos de las neuronas. Este proceso se realiza una sola vez al principio del programa.
Puede tardar hasta 60 segundos dependiendo del modelo.

```
def Inicializacion(configpath):
    start=time.process_time()
    print("[INFO]Inicializando camara...")
    Camara = picam.PiCamera()
    time.sleep(2)
    Camara.resolution=(320,240)
    Camara.shutter_speed = Camara.exposure_speed
    Camara.exposure_mode = 'off'
    g = Camara.awb_gains
    Camara.awb_mode = 'off'
    Camara.awb_gains = g
    raw=PiRGBArray(Camara,size=(320,240))
    time.sleep(2)
    stream=Camara.capture_continuous(raw,format="bgr",use_video_port=True)
    print("[INFO]Inicializando modelo...")
    Modelo,labels = Lector_modelo_dnn(configpath)
    end=time.process_time()
    print("[INFO]Fin inicializacion... tiempo={:.2f}".format(end-start))
    return Modelo,labels,stream,raw
```

5.3.2 Adquisición de imágenes

Se ha implementado un stream continuo de imágenes sin codificar provenientes de la cámara, esto agiliza mucho el proceso de adquisición de imágenes dentro del bucle.

```
def Get_frame(camera,raw):
    camera.capture(raw,format="bgr")
    image=raw.array
    image=np.array(image,dtype="float")/255.0
    cv2.imshow("capt init",image)
    return image
```

5.3.3 Preprocesamiento

Las imágenes se capturan a la misma resolución que se generaron para el entrenamiento y reciben el mismo preprocesado para garantizar que se obtienen los mismos resultados.

```
def tratar_imagen(imagen):
    #diversas operaciones para adaptar la imagen a lo quenecesita la red
    imagen=cv2.resize(imagen,(64,64))
    _imagen=imagen.reshape((1,imagen.shape[0],imagen.shape[1],imagen.shape[2]))
    cv2.imshow("resized",imagen)
    return _imagen
```

5.3.4 Evaluación

La imagen obtenida se hace pasar a través del modelo, y al vector de salida se le aplica el intérprete generado durante el entrenamiento, con lo que se obtiene la lista de las etiquetas posibles con cada una de las probabilidades, de la cual se escoge la que tiene el valor más alto y se devuelve como resultado.

```
def evaluar_imagen(imag_tratada, Modelo):
    predicciones=Modelo.predict(imag_tratada,batch_size=None,verbose=1)[0]
    return predicciones

def analizar_resultado(resultado, Labels):
    idx=np.argmax(resultado)
    label=Labels.classes_[idx]
    return label, idx
```

5.3.5 Muestra de Resultados

Al live feed que se muestra por pantalla se le sobrepone un texto que contiene la etiqueta con la mayor probabilidad, el valor exacto de probabilidad obtenida (en %) y los FPS que el bucle está consiguiendo

Además se imprime por la consola la misma información para poder obtener un log de toda la ejecución.

```
def mostrar_resultado(result, label, idx, imagen,fps):
    lab="{}: {:.2f}% FPS={:.2f}".format(label,result[idx]*100,fps)
    cv2.putText(imagen,lab,(10,25),cv2.FONT_HERSHEY_SIMPLEX,0.7,(0,255,0),2)
    cv2.imshow("resultado",imagen)
    print(lab)
    print("\n")
```

6. RESULTADOS

Se han generado 3 modelos diferentes basados en Vggnet aunque adaptados al caso particular, modificando valores y añadiendo y quitando capas a base de prueba y error además de una versión modificada de Alexnet a la que se le ha añadido Dropout para evitar el sobreentrenamiento.

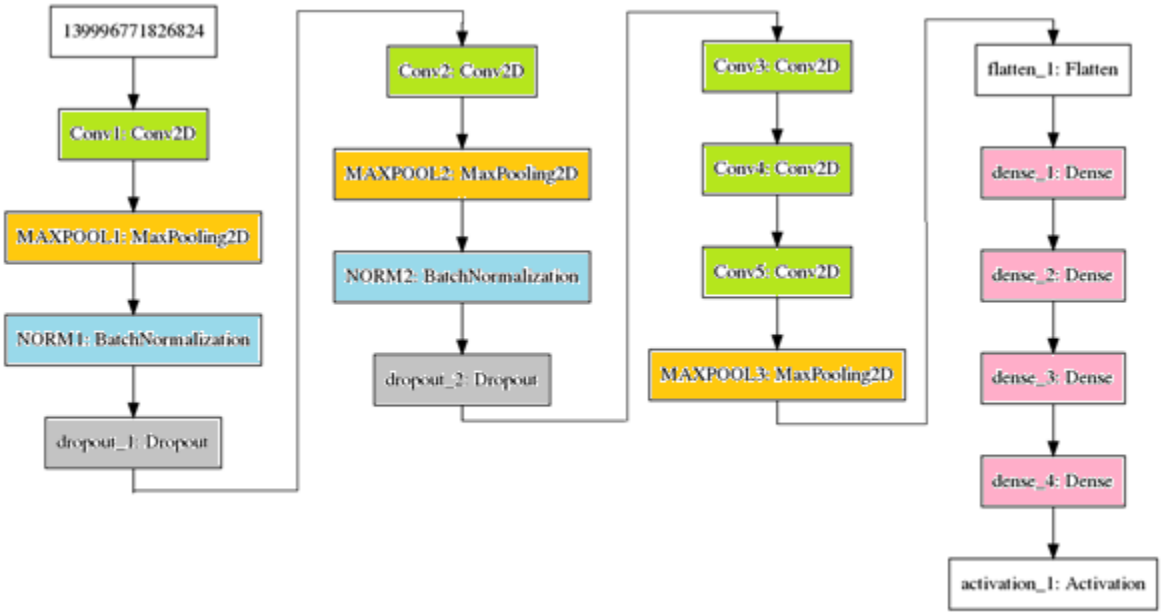
El siguiente paso es comprobar los resultados tanto en la Workstation como en la Raspberry pi

6.1 WORKSTATION

6.1.1 Alexnet

El modelo Alex net (Ilustración 19) fue el primero en aplicar capas Convolucionales y ganar premios con ellas. Supuso una revolución y a partir de entonces se hizo un requerimiento usar CNN en competición. Todo el modelo esta implementado en el Anexo 3

Ilustración 19 Estructura interna Modelo Alexnet



La red, primero pasa por dos grupos iguales de capas, estos grupos están formados por una capa convolucional con un filtro 10x10 de profundidad 96 con stride 4.
Esto significa que esta primera capa recorre la imagen cogiendo grupo s de 100 pixeles (10x10). De estos, extrae 96 características, y a continuación se desplaza 4 pixeles a la derecha y repite la operación hasta terminar la imagen.

A continuación MaxPooling se encarga de resumir toda esta información, en este caso en un filtro de 3x3 recorre la imagen dejando solo el valor más alto de los 9 actualmente en el filtro.
Después se normalizan las 96 características por pixel que se han extraído anteriormente evitando que los valores de algunas neuronas se eleven muy por encima del resto de valores.
La capa de Dropout “apaga” en cada iteración del entrenamiento neuronas aleatorias, en este caso cada una de las neuronas a la entrada de la capa tiene un 15% de probabilidades de apagarse.

El siguiente grupo de capas hace lo mismo lo único que cambia son tamaños de filtros y la profundidad de los mismos.
En este caso el filtro es 5x5 y 256 valores de profundidad. Esta gran cantidad de información extraída es lo que da nombre al método de Deep Neural Network

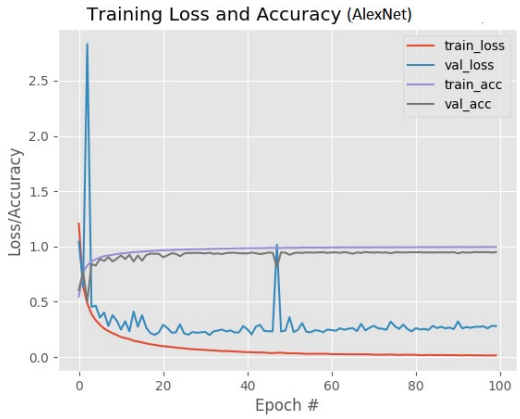
Las 3 capas seguidas Convolucionales buscan obtener aún más información en bruto, con esta información capas futuras podrán decidir la clase correspondiente a la imagen que ha entrado en un principio. En concreto, solo este grupo de capas suponen más de 3500000 parámetros diferentes que controlan su comportamiento cuando se ha obtenido toda esta información en bruto de max pooling reduce aún más el ancho y alto de la imagen y la profundidad final es de 256 características.

Como último paso, Flatten convierte ese volumen de características (6x6x256) en un vector al que conecta una capa de 4096 neuronas. Cada una de esas neuronas está conectada a todas las características del vector. Y a la salida de esta capa otra vez se repiten otras 4096 neuronas conectadas entre sí.
La siguiente capa Densa, busca reducir el número de características a 1000 y la siguiente lo reduce al número de clases diferentes que puede encontrar el problema, en este caso serán 11.

La última capa de activación Softmax se encarga de interpretar el vector de salida de estas últimas 11 neuronas como un vector de probabilidades, lo que genera, para cada una de las clases disponibles, la probabilidad de que la imagen, que entró y ahora se ha transformado en 11 valores, pertenezca a cada una de las clases.

El resultado del entrenamiento y su progreso se muestra a continuación (Ilustración 20):

Ilustración 20 Datos de entrenamiento Alexnet



Tras hacer la validación sobre el modelo con datos vírgenes los datos obtenidos son los siguientes (Tabla 2):

Tabla 2 Resultados de evaluación modelo Alexnet

	precisión	recall	f1-score	support
NoValido	0.88	0.52	0.66	1109.00
cocacola	0.96	0.99	0.98	1097.00
corazon	0.95	0.99	0.97	1079.00
dedo	0.96	0.99	0.97	1153.00
fresa	0.94	0.99	0.97	1100.00
jamon	0.93	0.98	0.95	1079.00
mora	0.97	0.99	0.98	1145.00
nube	0.92	0.97	0.94	1061.00
platano	0.97	1.00	0.98	1090.00
taco	0.95	1.00	0.97	1123.00
tarta	0.92	0.97	0.94	1064.00
avg / total	0.94	0.94	0.94	12100.00

El tiempo total de entrenamiento fue de 1,33 horas

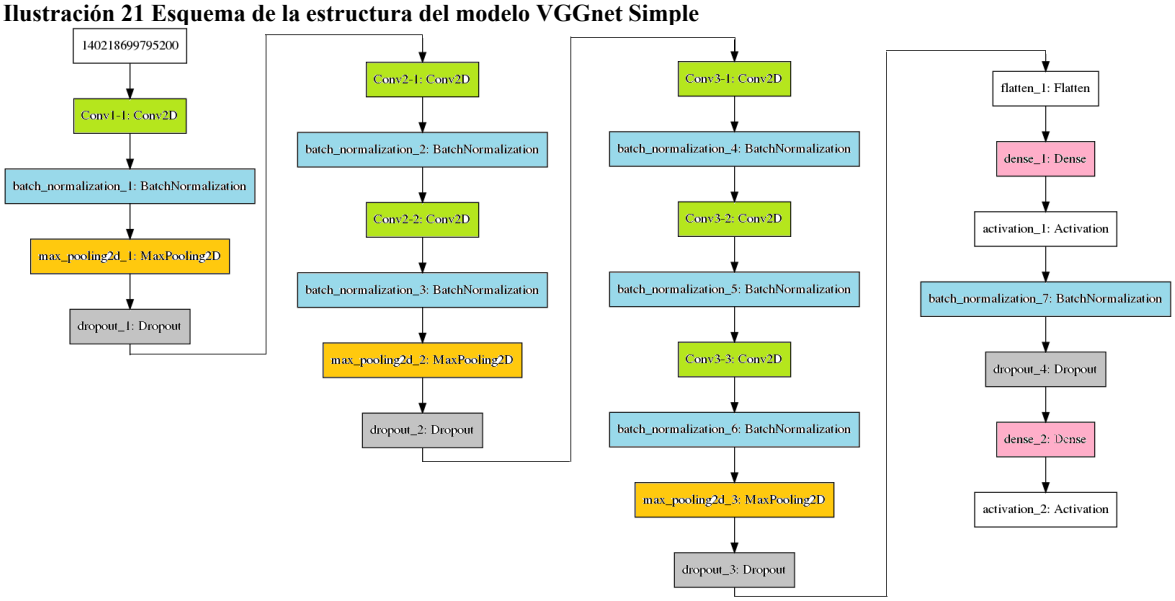
6.1.2 VGGnet

Todo el modelo esta implementado en una función (Anexo5) para facilitar su uso en el código de entrenamiento.

VGGnet se centra en utilizar filtros más pequeños en sus capas, (3x3), pero de mayor profundidad, esto reduce la cantidad de parámetros necesarios pero hace el aprendizaje más lento y requiere que el incremento de aprendizaje sea más bajo para aumentar la probabilidad de llegar a una solución como en el caso anterior VGGnet se centra en repetición de paquetes similares formados por una

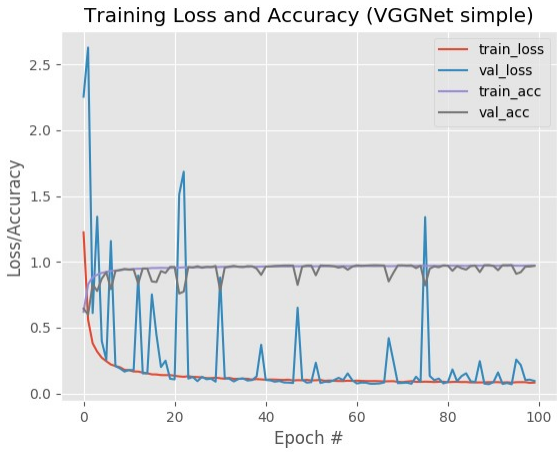
capa convolucinal, una capa de normalización y otra de max pooling, estos grupos de capas están conectados con capas de dropout
En el caso de VGGnet cada grupo de capas incorpora una capa de convolucion con su correspondiente normalización. El último grupo empieza transformando el volumen de características a un vector, que es conectado a una capa Densa tras normalización y Dropout se conecta a la capa densa de 11 características con activación Softmax.

Ilustración 21 Esquema de la estructura del modelo VGGnet Simple



Tras 150 epochs de entrenamiento se ha obtenido la siguiente información (Ilustración 22) que parece mostrar sería necesario más tiempo de entrenamiento.

Ilustración 22 Resultado entrenamiento de una VGGnet simple, durante 150 epoch



Al evaluar el modelo con los datos vírgenes se han obtenido los siguientes datos (Tabla 3)

Tabla 3 Resultado evaluación del modelo VGGnet simple

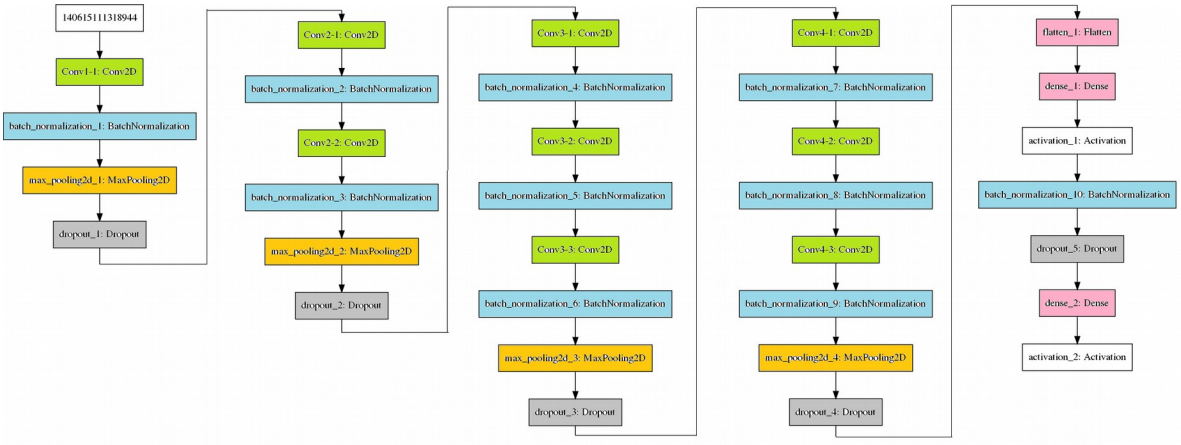
	precisión	recall	f1-score	support
NoValido	0.85	0.76	0.8	1109
cocacola	0.98	1	0.99	1097
corazon	0.98	1	0.99	1079
dedo	0.98	0.98	0.98	1153
fresa	0.98	1	0.99	1100
jamon	0.96	1	0.98	1079
mora	0.97	1	0.99	1145
nube	0.98	0.98	0.98	1061
platano	0.98	1	0.99	1090
taco	0.98	0.97	0.98	1123
tarta	0.96	0.94	0.95	1064
avg / total	0.96	0.97	0.97	12100

El tiempo total de entrenamiento fue de 2,41 horas

6.1.3 VGGnet ampliada

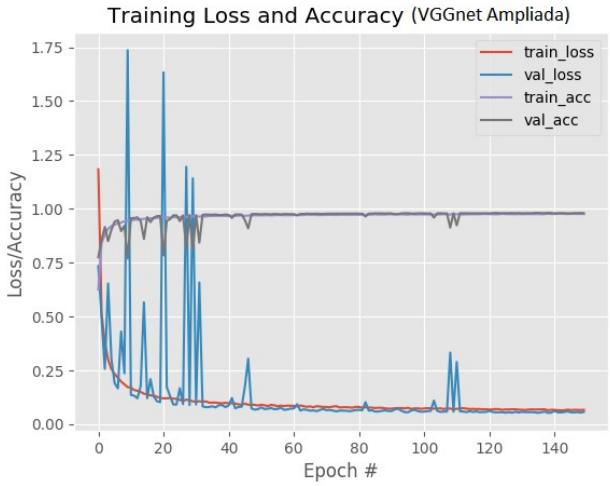
Esta red es un intento de mejora de la red VGGnet anterior, a la que se le ha duplicado el último grupo convolucional (Ilustración 23) aumentando la profundidad de los filtros con la intención de aumentar la cantidad de información obtenida, y para compensar se han implementado 2 capas extra de capas Densas antes de la salida.

Ilustración 23 Estructura del modelo VGGnet Ampliada



Tras el entrenamiento, los datos recogidos (Ilustración 24) muestran un mejor resultado que el modelo anterior.

Ilustración 24 Resultado del entrenamiento para una VGGnet ampliada durante 150 epochs



Al evaluar el modelo con los datos de validación no usados, se han obtenido los siguientes resultados (Tabla 4)

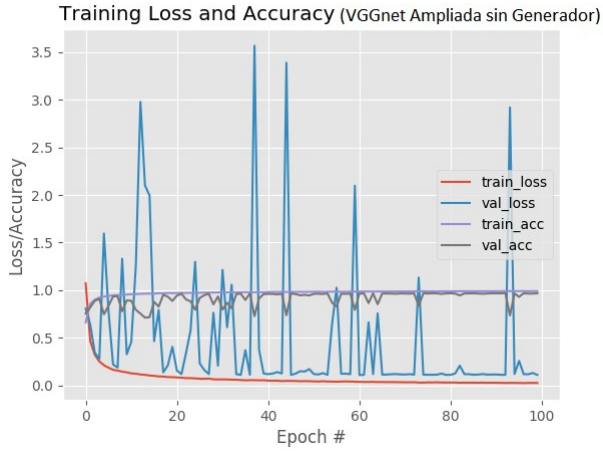
Tabla 4 resultados Validación VGGnet Ampliada

	precisión	recall	f1-score	support
NoValido	0.99	0.77	0.87	1109.00
cocacola	0.98	1.00	0.99	1097.00
corazon	0.98	1.00	0.99	1079.00
dedo	0.97	1.00	0.99	1153.00
fresa	0.98	1.00	0.99	1100.00
jamon	0.97	1.00	0.99	1079.00
mora	0.98	1.00	0.99	1145.00
nube	0.98	1.00	0.99	1061.00
platano	0.98	1.00	0.99	1090.00
taco	0.98	1.00	0.99	1123.00
tarta	0.97	1.00	0.98	1064.00
avg / total	0.98	0.98	0.98	12100.00

6.1.4 VGGnet ampliada sin generador de datos

La estructura de este modelo es exactamente la misma que en el caso anterior, pero en este caso durante el entrenamiento en vez de usar la opción de generar más imágenes a partir de las existentes, se ha optado por utilizar única y exclusivamente las generadas en un principio por el programa GenXuxes.
Al terminar el entrenamiento queda claro que aunque los modelos son los mismos los resultados (Ilustración 25) pueden variar enormemente por pequeños detalles como estos

Ilustración 25 Resultados del entrenamiento sin generador de datos del modelo VGGnet ampliada



Los datos teóricos obtenidos al someter al modelo a datos vírgenes (Tabla 5) no varían en gran medida del caso anterior, pero si se muestran ligeramente inferiores.

Tabla 5 resultado de validación del modelo VGGnet Ampliada sin generador

	precisión	recall	f1-score	support
NoValido	0.93	0.71	0.81	1109
cocacola	0.97	1	0.98	1097
corazon	0.97	1	0.98	1079
dedo	0.97	0.99	0.98	1153
fresa	0.97	1	0.98	1100
jamon	0.96	1	0.98	1079
mora	0.96	1	0.98	1145
nube	0.98	0.99	0.99	1061
platano	0.98	1	0.99	1090
taco	0.99	0.98	0.98	1123
tarta	0.97	1	0.98	1064
avg / total	0.97	0.97	0.97	12100

Una vez obtenido todos los resultados de validación el modelo más prometedor es el basado en VGGnet ampliada, además los resultados de las diferentes versiones de VGGnet en comparación con AlexNet muestra la ventaja de utilizar capas más profundas en vez de filtros más grandes.

6.2 SISTEMA EMPOTRADO

Una vez entrenados en la Workstation se ha de comprobar el rendimiento de los modelos en el sistema empotrado.
El programa PIendTFM.py (Anexo 7) se encarga tanto de la adquisición de las imágenes como de evaluar los modelos como ha sido explicado en el punto 5.3. En este caso nos fijaremos en la frecuencia máxima a la que el programa es capaz de analizar las imágenes, y el resultado que devuelve ya que será el elemento determinante de la viabilidad del sistema.

Para evaluar el comportamiento del sistema empotrado se va a tener en cuenta tanto el tiempo de evaluación, como los resultados que da. Para ello se van a hacer 5 pruebas de cada una de las 11 clases, que consistirá en depositar dos chuches del mismo tipo, dentro del alcance de la cámara, con luz y fondo controlados. Los objetivos se recogerán y recolocaran de forma aleatoria.
En el caso de la clase NoValido, una prueba será sin ningún objeto, 2 con un solo objeto suelto y las últimas 2 con alguna combinación aleatoria de 2 objetos.
Los resultados se plasmaran en una tabla con las clases presentadas en ambos ejes, en el horizontal, la clase que se está testeando y en el vertical la clase que el modelo ha devuelto como resultado. En consecuencia el resultado ideal es que todos los valores estén concentrados en la diagonal.

6.2.1 AlexNet

El Resultado de las pruebas realizadas con este modelo (Tabla 6), muestra una gran concentración de respuestas tanto en la clase fresa como taco, pero se ha comprobado que la clase mora la reconoce con gran precisión.

Tabla 6 Tabla de distribución de resultados.

Matriz de distribución de los resultados.	NoValido	cocacola	corazon	dedo	fresa	jamon	mora	nube	platano	taco	tarta
NoValido						1		1			1
cocacola											
corazon											
dedo											
fresa	3	2	1	2	5	2		1	1	1	2
jamon											
mora							5				
nube											
platano											
taco	2	3	4	3		2		3	4	4	2
tarta											

En cuanto al rendimiento del modelo, resulta bastante lento en su ejecución (Ilustración 26). Aunque algunos resultados eran positivos (Ilustración 27) la mayoría se debían a que eran la misma clase que el propio modelo identificaba erróneamente en el resto de los casos.

Ilustración 26 Extracción de la consola durante la evaluación del modelo AlexNet

```
▫1/1 [==============================] - 0s 434ms/step
taco: 100.00% FPS=1.59

▫1/1 [==============================] - 0s 430ms/step
taco: 100.00% FPS=1.58

▫1/1 [==============================] - 0s 422ms/step
taco: 100.00% FPS=1.60

▫1/1 [==============================] - 0s 421ms/step
taco: 100.00% FPS=1.60

▫1/1 [==============================] - 0s 422ms/step
taco: 100.00% FPS=1.59
```

Ilustración 27 Ejemplos de pruebas realizadas

6.2.2 VGGnet simple

El resultado en este caso es incluso pero que en el anterior (Tabla 7), ya que se hace evidente la tendencia a dar el resultado entre 2 clases independientemente de la información de entrada.

Tabla 7 Tabla de distribución de resultados

Matriz de distribución de los resultados.	NoValido	cocacola	corazon	dedo	fresa	jamon	mora	nube	platano	taco	tarta
NoValido											
cocacola											
corazon											
dedo	2	3	4	4		5		2		5	1
fresa											
jamon											
mora	3	2	1	1	5		5	3	5		4
nube											
platano											
taco											
tarta											

En cuestión de rendimiento sin embargo este modelo evalúa la imagen casi el doble de rápido que la anterior (Ilustración 28), lo que la hace más adecuada para un entorno de producción más rápido.

Ilustración 28 Extracción de la consola durante la evaluación del modelo VGGnet simple

```
▫1/1 [==============================] - 0s 266ms/step
taco: 100.00% FPS=1.60

▫1/1 [==============================] - 0s 269ms/step
taco: 100.00% FPS=1.63

▫1/1 [==============================] - 0s 261ms/step
taco: 100.00% FPS=1.63

▫1/1 [==============================] - 0s 245ms/step
taco: 100.00% FPS=1.64

▫1/1 [==============================] - 0s 253ms/step
taco: 100.00% FPS=1.66

▫1/1 [==============================] - 0s 247ms/step
taco: 100.00% FPS=1.63
```

Se comprueba con todas las clases diferentes posiciones de los objetos, como superposiciones (Ilustración 29) o elementes parcialmente fuera de cuadro.

Ilustración 29 ejemplo muestra de clase NoValido

6.2.3 VGGnet aumentada

En el caso de este modelo debido a los resultados del historial del entrenamiento se han realizado 2 sets de pruebas, la primera(Tabla 8) con el modelo entrenado hasta 150 epochs, y una segunda, escogida con el criterio de evitar al máximo el sobreentrenamiento del modelo, obteniendo la mayor precisión posible(Tabla 9) entrenándolo hasta las 60 epochs.

Tabla 8 Matriz de resultados con 150 epochs

Matriz de distribución de los resultados.	NoValido	cocacola	corazon	dedo	fresa	jamon	mora	nube	platano	taco	tarta
NoValido	3	4	4	3	4	4	2		4	4	1
cocacola											
corazon											
dedo											
fresa											
jamon											
mora											
nube											
platano											
taco	2	1	1	2	1	1	3	5	1	1	4
tarta											

Como se puede apreciar observando los resultados poco ha importado el cambio en el número de epochs, sí que se aprecian los resultados distribuidos más entre dos clases, pero las mediciones siguen siendo erróneas.

Tabla 9Matriz de resultados con 60 epochs

Matriz de distribución de los resultados.	NoValido	cocacola	corazon	dedo	fresa	jamon	mora	nube	platano	taco	tarta
NoValido	5	5	4	5	5	4	5	5	5	5	5
cocacola											
corazon											
dedo											
fresa											
jamon											
mora											
nube											
platano											
taco			1			1					
tarta											

El tiempo de respuesta del modelo (Ilustración 30) es ligeramente superior que el de VGGnet simple, como era de esperar ya que el modelo es más complejo.

Ilustración 30 Extracción de la consola durante la evaluación del modelo VGGnet aumentado

```
▫1/1 [==============================] - 0s 297ms/step
NoValido: 100.00% FPS=1.18

▫1/1 [==============================] - 0s 291ms/step
NoValido: 100.00% FPS=1.20

▫1/1 [==============================] - 0s 311ms/step
NoValido: 100.00% FPS=1.21

▫1/1 [==============================] - 0s 293ms/step
NoValido: 100.00% FPS=1.23

▫1/1 [==============================] - 0s 294ms/step
NoValido: 100.00% FPS=1.21

▫1/1 [==============================] - 0s 384ms/step
NoValido: 100.00% FPS=1.17
```

El tipo de pruebas fue igual que en el resto de modelos.

6.2.4 VGGnet aumentada sin generador

Los resultados obtenidos(Tabla 10) muestran una única clase resultante independientemente de los objetos utilizados.

Tabla 10 Matriz de resultados para modelo entrenado sin generador de datos

Matriz de distribución de los resultados.	NoValido	cocacola	corazon	dedo	fresa	jamon	mora	nube	platano	taco	tarta
NoValido											
cocacola											
corazon											
dedo											
fresa											
jamon											
mora											
nube											
platano											
taco	5	5	5	5	5	5	5	5	5	5	5
tarta											

El tiempo de respuesta (Ilustración 31) es idéntico al anterior ya que el modelo es el mismo y solo cambia el método de entrenamiento, que siendo el mismo número de epochs en los 2 casos se demuestra un factor relevante.

Ilustración 31 Extracción de la consola del modelo VGGnet aumentado sin generador

```
1/1 [==============================] - 0s 297ms/step
taco: 100.00% FPS=1.21

1/1 [==============================] - 0s 311ms/step
taco: 100.00% FPS=1.21

1/1 [==============================] - 0s 310ms/step
taco: 100.00% FPS=1.21

1/1 [==============================] - 0s 299ms/step
taco: 100.00% FPS=1.21

1/1 [==============================] - 0s 305ms/step
taco: 100.00% FPS=1.20
```

7. ANALISIS Y DISCUSION DE RESULTADOS

A continuación se analizaran los resultados obtenidos con cada uno de los modelos generados y evaluados y se extraerá información a partir de los mismos.

7.1 ALEXNET

Los datos obtenidos durante el entrenamiento nos aportan mucha información de cómo el modelo debería de comportarse una vez desplegado en la Raspberry pi.

7.1.1 Entrenamiento

Se puede apreciar por la gráfica del entrenamiento (Ilustración 19) que para las condiciones de entrenamiento aplicadas el número óptimo de epochs es entre 20 y 30, y que a partir de este valor el modelo deja de comportarse de manera correcta. Aunque el valor mínimo de error de testeo medido es de 0.2 los resultados con las imágenes de validación (Tabla 2) indica un buen ratio de reconocimiento y recall de todas las clases menos NoValido, sobre todo sufre en el resultado de recall lo que indica que hay una gran cantidad de falsos negativos.
Cabe destacar que el tiempo de entrenamiento es bastante más bajo que el del resto de modelos. Esto se puede deber a la menor cantidad de capas que acumula este tipo de modelo.

7.1.2 Despliegue

Al desplegar el modelo se hace patente que al cambiar los datos de entrada, la información obtenida del historial de entrenamiento no influye, ya que aunque este modelo es el que peor resultados tiene en el historial, es el único que ha conseguido identificar un objeto de forma precisa y repetible.
Lo siguiente relevante es el tiempo en el que el modelo es evaluado, para ello tenemos 2 mediciones, la primera mostrada por la línea de comandos la devuelve la propia función de evaluación del modelo en forma de ms/stepp, siendo que en estos casos el step siempre es 1, indica el tiempo en ms que ha tardado el modelo en analizar la imagen. Además de forma externa a la librería del modelo se ha medido el tiempo entre antes de empezar la evaluación y después de analizar el resultado, mostrado en Frames Per Second (FPS). Para realizar estas mediciones de tiempo se ha esperado a que el valor se estabilice para que no interfieran factores externos a la prueba realizada.
En este caso los valores discrepan ligeramente hay que tener en cuenta que el valor de FPS incluye un mayor número de operaciones.

7.2 VGGNET SIMPLE

7.2.1 Entrenamiento

Durante el entrenamiento el rendimiento de este tipo de redes ha se han obtenido datos (Ilustración 21), que indican que el resultado del entrenamiento es poco fiable, con valores muy dispares de error de testeo entre epochs, aunque el valor mínimo es inferior al mínimo del modelo AlexNet, en este caso rozando el 0,1.
Sin embargo la inestabilidad de este resultado puede indicar que el modelo se está sobre entrenando o que los pasos de entrenamiento son demasiado grandes y no permiten llegar a una solución mejor.
Sin embargo a partir de la información obtenida a partir de la evaluación de las imágenes sin usar, muestra (Tabla 3) un aumento considerable en la precisión y el recall sobre todo en la clase NoValido que aumenta de 52% al 76% de recall.

7.2.2 Despliegue

Al desplegar el modelo en el sistema empotrado, se ha comprobado que no se consiguen resultados correctos a partir de las imágenes obtenidas por la Raspberry pi (Ilustración 27). Se puede apreciar que este modelo utiliza menos parámetros ya que su tiempo de evaluación se ha reducido casi a la mitad respecto a AlexNet, sin embargo los FPS no han subido en proporción.

7.3 VGGNET AUMENTADA

7.3.1 Entrenamiento

Durante el entrenamiento el rendimiento de esta red se han obtenido datos (Ilustración 23) con un valor final semejante a la versión simple de VGGnet, pero esta vez con una respuesta mucho más estable que en el caso anterior. Además los resultados obtenidos con las imágenes vírgenes son también prometedores (Tabla 4), los cuales muestran un valor de recall perfecto en todas las clases menos NoValido que mantiene un 77% y la media de precisión de todas las clases es 98%. Estos resultados son altamente prometedores y los mejores de momento.

7.3.2 Despliegue

Con este modelo se han desplegado 2 versiones, cada una entrenada un numero diferente de epochs, aunque ambas con resultados muy desfavorables, además se ha mostrado que un número menor de epochs tiende a dar resultados más centrados en una sola clase que un modelo entrenado durante más tiempo aunque no haya tenido mejora en la precisión durante ese tiempo.
En lo referente a la velocidad de análisis y FPS se puede apreciar (Ilustración 29) un ligero aumento del tiempo necesario para evaluar el modelo respecto a su versión simplificada, y su consiguiente descenso de FPS.

7.4 VGGNET AUMENTADA SIN GENERADOR

7.4.1 Entrenamiento

Al ser el mismo tipo de modelo que el caso anterior cabría esperar un comportamiento similar ante una entrada de datos similar, aunque como se puede observar por los resultados del entrenamiento (Ilustración 24) los resultados se parecen más al comportamiento de la versión simple de VGGnet, además de que el valor mínimo no se acerca al del caso anterior.

El resultado del análisis con imágenes vírgenes (Tabla5) se corresponde de nuevo más con la versión simplificada que la obtenida en el caso anterior.

7.4.2 Despliegue

Todo esto junto con un tiempo de evaluación (Ilustración 31) igual o mayor que el del modelo VGGnet ampliada, hace que este modelo haya obtenido lo malo de los 2 modelos anteriores.

Además ha mostrado el peor resultado en la matriz de resultados de todos los modelos evaluados.

8. CONCLUSIONES

- Se ha comprobado una disparidad entre los resultados teóricos durante el entrenamiento y los resultados obtenidos al desplegar los modelos en el sistema empotrado. Como para la validación de los modelos se ha usado información que no había pasado antes por el modelo, la causa de este cambio de comportamiento se relega a que las imágenes tomadas por la Raspberry, y las generadas por computador para el entrenamiento son demasiado diferentes como para que la red las detecte correctamente.
- Con el trabajo realizado se ha comprobado la importancia de un buen entorno que te permita actuar con flexibilidad y realizar cambios de forma sencilla para poder hacer la mayor cantidad de pruebas posible
- La Raspberry, según para que aplicaciones del ámbito industrial podría ser un buen sistema de visión artificial basado en redes neuronales. El principal factor limitante es la velocidad a la que es capaz de analizar las imágenes que la expulsaría de cualquier proceso industrial en el que el flujo de elementos a identificar fuese más rápido de 500ms/pieza.
- Por un precio de 57 € se obtiene un sistema que dentro de sus límites es capaz de funcionar como sistema de visión. Esto lo hace apto para su uso por empresas pequeñas con un presupuesto reducido.
- La Workstation necesaria para realizar el entrenamiento aunque más cara es más polivalente y el gasto depende de la prisa que se tenga a la hora de entrenar nuevos modelos, o actualizar modelos antiguos con nuevas imágenes o clases.
- SE ha comprobado que el uso del generador de datos incluido en las APIs de Redes Neuronales es completamente necesario. Por lo tanto no es necesario una cantidad tan abrumadora de imágenes generadas.

9. RECOMENDACIONES Y TRABAJOS FUTUROS

- Se pueden usar imágenes generadas con ordenador siempre y cuando estas se basen en imágenes tomadas con el mismo dispositivo con el que se va a realizar la adquisición de datos durante el despliegue del modelo.
- No son necesarias tantas imágenes como se han generado durante este proyecto ya que el generador incluido en las librerías se podría considerar necesario a partir de los resultados obtenidos en los ensayos.
- Es conveniente habilitar un entorno de adquisición de imágenes adecuado y acorde con el entorno de aplicación del sistema empotrado. Ya es un problema suficientemente complicado de base como para añadirle dificultad extra mediante la indeterminación de las condiciones del entorno
- Como trabajo futuro volver a entrenar las redes, esta vez con imágenes obtenidas con la Raspberry pi, puesto que tienen una mayor probabilidad de dar un resultado satisfactorio.
- También sería interesante aumentar el tipo de modelos y descubrir el límite de la Raspberry en cuanto a número de capas y complejidad máximas que soporta.
- Hacer hincapié en la importancia de unir antes en el desarrollo el proceso de entrenamiento y el de adquisición de datos definitivo. Preferiblemente usar únicamente imágenes obtenidas directamente de la cámara y que sea el generador de imágenes del entrenamiento el que aumente el número a partir de una base.

10. TRABAJOS CITADOS

Fei-Fei Li, J. J. (2 de 5 de 2017). Recuperado el 20 de 06 de 2018, de Stanford university: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf

john Lawrence, J. M. (05 de 05 de 2017). *Bronx comunity College.* Recuperado el 15 de 09 de 2018, de https://pdfs.semanticscholar.org/9492/86e2522c110c4b2976121ede6a8a7441efd5.pdf?_ga=2.22496315.1064281759.1536980705-805299376.1536980705

11. ANEXOS

11.1 ANEXO 1 GENXUXES

```
import argparse
import os.path
import random
from PIL import Image, ImageDraw
from tfmaux import *


def convert(size, box):
    dw = 1./size[0]
    dh = 1./size[1]
    x = (box[0] + box[1])/2.0
    y = (box[2] + box[3])/2.0
    w = box[1] - box[0]
    h = box[3] - box[2]
    x = x*dw
    w = w*dw
    y = y*dh
    h = h*dh
    return (x,y,w,h)


# entrada de datos,
# xuxe1
# xuxe2
# Número imagenes
ap = argparse.ArgumentParser()
ap.add_argument("-i1", "--image1", required=True, help="name of xuxe kind")
ap.add_argument("-i2", "--image2", required=False, help="name of second xuxe kind,
optional")
ap.add_argument("-n", "--number", required=True, help="Number of images resulting")
ap.add_argument("-d", "--outdir", required=False, help="output directory, default is
Results")
ap.add_argument("-m", "--meta", required=True, help="determines wether the program generates
metadata for each image or not", default= True)
args = vars(ap.parse_args())

if args["outdir"] is None:
    outdir = "Results/"
else:
    outdir = args["outdir"] + "/"

# variables auxiliares
n1 = 0
n2 = 0
Tags = 1
BB2 = 1
BB2s=1
numeroFondos= 4
```

```
res=[320,240]
# cuenta el número de imagenes en el directorio
for root1, dirs1, files1 in os.walk("Xuxes/" + args["image1"]):
    for f in files1:
        if f.endswith(".png"):
            n1 = n1 + 1

# cuenta el número de imagenes en el directorio
if args["image2"] is not None:
    Tags = 2
    for root2, dirs2, files2 in os.walk("Xuxes/" + args["image2"]):
        for f in files2:
            if f.endswith(".png"):
                n2 = n2 + 1

# bucle que se repite segun el numero especificado
for i in range(int(args["number"])):
    # for i in range(4):
    pathf = "Xuxes/fondos/" + "fondo" + str(random.randint(1, numeroFondos)) + ".jpg"
    #carga el fondo
    fondo = Image.open(pathf)
    #reescala el fondo al tamaño de la foto de la Raspberrypi
    fondo.thumbnail(res)


    # cargar xuxe 1 aleatoria de las existenetes en el directorio
    path1 = "Xuxes/" + args["image1"] + "/" + args["image1"] + str(random.randint(1, n1)) +
".png"
    xuxe1 = Image.open(path1)
    xuxe1.load()
    # reescala la xuxe a un tamaño aleatorio entre un minimo y la mitad de la resolucion del
fondo,
    #la idea es variar el tamaño de los objetos a reconocer
    rand_res=[random.randint(res[0]/4,res[0]/2),random.randint(res[1]/4,res[1]/2)]
    xuxe1.thumbnail((rand_res[0], rand_res[1]))
    # xuxe1.show()
    xuxe1 = xuxe1.rotate(random.randint(0, 360))  # se rota de forma aleatoria
    BB1 = xuxe1.getbbox()
    #test=ImageDraw.Draw(xuxe1)
    #test.rectangle(BB1,outline='red')
    # xuxe1.show()
    # print(str(BB1))

    # lo mismo para xuxe2 en caso de existir nombre
    if args["image2"] is not None:
        path2 = "Xuxes/" + args["image2"] + "/" + args["image2"] + str(random.randint(1,
n2)) + ".png"
        xuxe2 = Image.open(path2)
        # xuxe2.show()
        rand_res = [random.randint(res[0]/4, res[0] / 2), random.randint(res[1]/4, res[1] /
2)]
        xuxe2.thumbnail((rand_res[0], rand_res[1]))
        # xuxe2.show()
```

```
        xuxe2 = xuxe2.rotate(random.randint(0, 360))  # se rota de forma aleatoria
        BB2 = xuxe2.getbbox()
        # xuxe2.show()
        # print(path1)

    # una vez obtenido ambas xuxes y el fondo hay que juntarlo
    position = (random.randint(0 - BB1[0], (fondo.width - BB1[2])),
                random.randint(0 - BB1[0], (fondo.height - BB1[3])))
    fondo.paste(xuxe1, position, xuxe1)
    b = (float(BB1[0])+position[0], float(BB1[2])+position[0], float(BB1[1])+ position[1],
float(BB1[3])+ position[1])
    BB1s =convert(res,b)
    # se repite para segunda xuxe si necesario
    if args["image2"] is not None:
        position = (random.randint(0 - BB2[0], (fondo.width - (BB2[2]))),
                    random.randint(0 - BB2[1], (fondo.height - (BB2[3]))))
        fondo.paste(xuxe2, position, xuxe2)
        b = (float(BB2[0]) + position[0], float(BB2[2]) + position[0], float(BB2[1]) +
position[1],float(BB2[3]) + position[1])
        BB2s = convert(res, b)

    #una vez pegadas las partes se guarda el resultado con el nombre de la xuxe generada +
un numero
    fondo.save(outdir + str(args["image1"]) + str(i) + ".jpg")
    fondo.close()

    if args["meta"] is True:
        f = open(outdir + str(args["image1"]) + str(i) + ".txt", 'w')
        if args["image2"] is not None:
            f.write(str(Tags) + ";" + str(args["image1"]) + ";" + BB1s + ";" +
str(args["image2"]) + ";" + str(BB2s))
        else:
            f.write(str(LabelSW.get(args["image1"])) + " " + str(BB1s[0])+" " +str(BB1s[1])
+" " +str(BB1s[2])+" " +str(BB1s[3]))
        f.close()
    # cv2.waitKey()
# print(fondo)
```

11.2 ANEXO 3 TRAINVGG.PY

```
#!/bin/bash -p


# python train_vgg.py --dataset directorioraizimagenes --model directorioparaelmodelo.model
--label-bin directorioparalaslabels.pickle --plot directorioparalagrafica.png

# permite guardar las figuras sin mostrarlas
import matplotlib
matplotlib.use("Agg")
from keras.backend.tensorflow_backend import set_session
from pyimagesearch.smallvggnet import SmallVGGNet
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report , confusion_matrix
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import SGD
from keras.utils import plot_model
from imutils import paths
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import argparse
import random
import pickle
import cv2
import os

# constructor del intérprete de parámetros
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
          help="path al dataset de imagenes")
ap.add_argument("-m", "--model", required=True,
          help="path para guardar el modelo")
ap.add_argument("-l", "--label-bin", required=True,
          help="path para guardar las labels")
ap.add_argument("-p", "--plot", required=True,
          help="path para mostrar el grafico de entrenamiento")
args = vars(ap.parse_args())

# inicializa las variables que almacenan las imagenes y sus correspondientes labels
print("[INFO] cargando imagenes...")
data = []
labels = []

# mezcla el dataset para evitar aprender secuencias
imagePaths = sorted(list(paths.list_images(args["dataset"])))
random.seed(42)
random.shuffle(imagePaths)

# preprocesamiento de todas las imagenes
for imagePath in imagePaths:
```

```
            # lee la imagen y la modifica a 64X64 para entrar en la red diseñada
            image = cv2.imread(imagePath)
            image = cv2.resize(image, (64, 64))
            data.append(image)

            # a partir del path de la imagen podemos averiguar la etiqueta correspondiente
            # se añade a la lista de labels
            label = imagePath.split(os.path.sep)[-2]
            labels.append(label)

# normalizamos la entrada
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)

# DIvision de los datos en 3 grupos 60/20/20%
(trainX, tempX, trainY, tempY) = train_test_split(data,
            labels, test_size=0.40, random_state=42)
(testX, valX, testY, valY) = train_test_split(tempX,
            tempY, test_size=0.50, random_state=42)

#convertimos el total de posibles clases en un vector de 1y 0
#este vector será el intérprete entre el mundo y la red.
lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)
valY = lb.transform(valY)

#descomentar estas líneas en caso de que
#la memoria de la tarjeta no sea suficiente
#config = tf.ConfigProto()
#config.gpu_options.allow_growth = True
#set_session(tf.Session(config=config))

# Aquí se inicializa el ampliador de datos para el entrenamiento,
#especificando el tipo de modificaciones que se pueden aplicar.
aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
            height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
            horizontal_flip=True, fill_mode="nearest")

# Se carga en memoria el modelo seleccionado.
model = SmallVGGNet.build(width=64, height=64, depth=3,
            classes=len(lb.classes_))

# se inicializa el paso de entrenamiento, el número de epochs deseado
# y el tamaño de cada batch
INIT_LR = 0.01
EPOCHS = 100
BS = 32

# SE inicializa el optimizador del modelo usado en el entrenamiento,
# asi como el modelo en si especificando la función de error.
print("[INFO] Entrenando red...")
opt = SGD(lr=INIT_LR, decay=INIT_LR / EPOCHS)
```

```
model.compile(loss="categorical_crossentropy", optimizer=opt,
          metrics=["accuracy"])
#se guarda el esquema del modelo entrenado
plot_model(model,to_file="/home/anmurlo/Resultados_modelos/esquema60extranogen.png")

# empieza el entrenamiento del modelo
H = model.fit(trainX, trainY, batch_size=BS,
          validation_data=(testX, testY), epochs=EPOCHS)

# una vez se termina el entrenamiento se guarda el modelo y el intérprete en el disco
#para poder mandarlos a la Raspberry
print("[INFO] Guardando modelo e interprete...")
model.save(args["model"])
f = open(args["label_bin"], "wb")
f.write(pickle.dumps(lb))
f.close()

# Una vez terminado el entrenamiento se procede a evaluar el resultado.
print("[INFO] evaluando modelo...")
predictions = model.predict(valX, batch_size=32)
print(classification_report(valY.argmax(axis=1),
          predictions.argmax(axis=1), target_names=lb.classes_))

# se genera y guarda la gráfica de error y precisión
# a partir del historial generado durante el entrenamiento
N = np.arange(0, EPOCHS)
plt.style.use("ggplot")
plt.figure()
plt.plot(N, H.history["loss"], label="train_loss")
plt.plot(N, H.history["val_loss"], label="val_loss")
plt.plot(N, H.history["acc"], label="train_acc")
plt.plot(N, H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy (SmallVGGNet)")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.savefig(args["plot"])
```

11.3 ANEXO4 ALEXNET

```
# import the necessary packages
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras import backend as K

class AlexNet:
          @staticmethod
          def build(width, height, depth, classes):
                    # initialize the model along with the input shape to be
                    # "channels last" and the channels dimension itself
                    model = Sequential()
                    inputShape = (height, width, depth)
                    chanDim = -1

                    # if we are using "channels first", update the input shape
                    # and channels dimension
                    if K.image_data_format() == "channels_first":
                              inputShape = (depth, height, width)
                              chanDim = 1

                    # CONV => RELU => POOL layer set
                    model.add(Conv2D(96, (10,10), padding="same",

          input_shape=inputShape,name="Conv1",activation="relu",strides=4))
                    model.add(MaxPooling2D(pool_size=(3, 3),name="MAXPOOL1",strides=2))
                    model.add(BatchNormalization(axis=chanDim,name="NORM1"))
                    model.add(Dropout(0.15))

                    model.add(Conv2D(256, (5,5), padding="same",name="Conv2",
activation="relu", strides=1))
                    model.add(MaxPooling2D(pool_size=(3, 3), name="MAXPOOL2", strides=2))
                    model.add(BatchNormalization(axis=chanDim, name="NORM2"))
                    model.add(Dropout(0.15))

                    model.add(Conv2D(384, (3, 3), padding="same", name="Conv3",
activation="relu", strides=1))
                    model.add(Conv2D(384, (3, 3), padding="same", name="Conv4",
activation="relu", strides=1))
                    model.add(Conv2D(256, (3, 3), padding="same", name="Conv5",
activation="relu", strides=1))
                    model.add(MaxPooling2D(pool_size=(3, 3), name="MAXPOOL3", strides=2))

                    model.add(Flatten())
                    model.add(Dense(4096))
                    model.add(Dense(4096))
```

```
model.add(Dense(1000))
# softmax classifier
model.add(Dense(classes))
model.add(Activation("softmax"))


# return the constructed network architecture
return model
```

11.4 ANEXO 5 VGGNET SIMPLE

```
# import the necessary packages
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras import backend as K

class VGGNetsimple:
	@staticmethod
	def build(width, height, depth, classes):
		# initialize the model along with the input shape to be
		# "channels last" and the channels dimension itself
		model = Sequential()
		inputShape = (height, width, depth)
		chanDim = -1

		# if we are using "channels first", update the input shape
		# and channels dimension
		if K.image_data_format() == "channels_first":
			inputShape = (depth, height, width)
			chanDim = 1

		# CONV => RELU => POOL layer set
		model.add(Conv2D(32, (3, 3), padding="same",
			input_shape=inputShape,name="Conv1-1",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
		model.add(MaxPooling2D(pool_size=(2, 2)))
		model.add(Dropout(0.15))

		# (CONV => RELU) * 2 => POOL layer set
		model.add(Conv2D(64, (3, 3), padding="same",name="Conv2-
1",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
		model.add(Conv2D(64, (3, 3), padding="same",name="Conv2-
2",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
		model.add(MaxPooling2D(pool_size=(2, 2)))
		model.add(Dropout(0.15))

		# (CONV => RELU) * 3 => POOL layer set
		model.add(Conv2D(128, (3, 3), padding="same",name="Conv3-
1",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
		model.add(Conv2D(128, (3, 3), padding="same",name="Conv3-
2",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
```

```
                    model.add(Conv2D(128, (3, 3), padding="same",name="Conv3-
3",activation="relu"))
                    model.add(BatchNormalization(axis=chanDim))
                    model.add(MaxPooling2D(pool_size=(2, 2)))
                    model.add(Dropout(0.15))

                    # first (and only) set of FC => RELU layers
                    model.add(Flatten())
                    model.add(Dense(512))
                    model.add(Activation("relu"))
                    model.add(BatchNormalization())
                    model.add(Dropout(0.25))

                    # softmax classifier
                    model.add(Dense(classes))
                    model.add(Activation("softmax"))

                    # return the constructed network architecture
                    return model
```

11.5 ANEXO 6 VGGNET AMPLIADA

```
# import the necessary packages
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras import backend as K

class VGGNetAmpliada:
	@staticmethod
	def build(width, height, depth, classes):
		# initialize the model along with the input shape to be
		# "channels last" and the channels dimension itself
		model = Sequential()
		inputShape = (height, width, depth)
		chanDim = -1

		# if we are using "channels first", update the input shape
		# and channels dimension
		if K.image_data_format() == "channels_first":
			inputShape = (depth, height, width)
			chanDim = 1

		# CONV => RELU => POOL layer set
		model.add(Conv2D(32, (3, 3), padding="same",
			input_shape=inputShape,name="Conv1-1",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
		model.add(MaxPooling2D(pool_size=(2, 2)))
		model.add(Dropout(0.15))

		# (CONV => RELU) * 2 => POOL layer set
		model.add(Conv2D(64, (3, 3), padding="same",name="Conv2-
1",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
		model.add(Conv2D(64, (3, 3), padding="same",name="Conv2-
2",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
		model.add(MaxPooling2D(pool_size=(2, 2)))
		model.add(Dropout(0.15))

		# (CONV => RELU) * 3 => POOL layer set
		model.add(Conv2D(128, (3, 3), padding="same",name="Conv3-
1",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
		model.add(Conv2D(128, (3, 3), padding="same",name="Conv3-
2",activation="relu"))
		model.add(BatchNormalization(axis=chanDim))
```

```
                    model.add(Conv2D(128, (3, 3), padding="same",name="Conv3-
3",activation="relu"))
                    model.add(BatchNormalization(axis=chanDim))
                    model.add(MaxPooling2D(pool_size=(2, 2)))
                    model.add(Dropout(0.15))

                    # (CONV => RELU) * 3 => POOL layer set
                    model.add(Conv2D(256, (3, 3), padding="same",name="Conv4-
1",activation="relu"))
                    model.add(BatchNormalization(axis=chanDim))
                    model.add(Conv2D(256, (3, 3), padding="same",name="Conv4-
2",activation="relu"))
                    model.add(BatchNormalization(axis=chanDim))
                    model.add(Conv2D(256, (3, 3), padding="same",name="Conv4-
3",activation="relu"))
                    model.add(BatchNormalization(axis=chanDim))
                    model.add(MaxPooling2D(pool_size=(2, 2)))
                    model.add(Dropout(0.15))

                    # first (and only) set of FC => RELU layers
                    model.add(Flatten())
                    model.add(Dense(512))
                    model.add(Activation("relu"))
                    model.add(BatchNormalization())
                    model.add(Dropout(0.25))



                    # softmax classifier
                    model.add(Dense(classes))
                    model.add(Activation("softmax"))

                    # return the constructed network architecture
                    return model
```

11.6 ANEXO 7 PIENDTFM

```
import picamera as picam
from picamera.array import PiRGBArray
from PIL import Image
import keras as K
import numpy as np
import pickle
import time
import cv2
import io
import os


def Inicializacion(configpath):
    start=time.process_time()
    print("[INFO]Inicializando camara...")
    Camara = picam.PiCamera()
    time.sleep(2)
    Camara.resolution=(320,240)
    Camara.shutter_speed = Camara.exposure_speed
    Camara.exposure_mode = 'off'
    g = Camara.awb_gains
    Camara.awb_mode = 'off'
    Camara.awb_gains = g
    raw=PiRGBArray(Camara,size=(320,240))
    time.sleep(2)
    stream=Camara.capture_continuous(raw,format="bgr",use_video_port=True)
    print("[INFO]Inicializando modelo...")
    Modelo,labels = Lector_modelo_dnn(configpath)
    end=time.process_time()
    print("[INFO]Fin inicializacion... tiempo={:.2f}".format(end-start))
    return Modelo,labels,stream,raw

def parar_captura(Stream,rawCapture):
    Stream.close()
    rawCapture.close()

def Lector_modelo_dnn(path):
    model= K.models.load_model(path+"/modelotest100extranogen.model")
    labels=pickle.loads(open(path+"/labelbin100extranogen.pickle",'rb').read())
    return model, labels

def Get_frame(camera,raw):
    camera.capture(raw,format="bgr")
    image=raw.array
    image=np.array(image,dtype="float")/255.0
    cv2.imshow("capt init",image)
    return image

def tratar_imagen(imagen):
    #diversas operaciones para adaptar la imagen a lo quenecesita la red
    imagen=cv2.resize(imagen,(64,64))
```

```
    _imagen=imagen.reshape((1,imagen.shape[0],imagen.shape[1],imagen.shape[2]))
    cv2.imshow("resized",imagen)
    return _imagen

def evaluar_imagen(imag_tratada, Modelo):
    predicciones=Modelo.predict(imag_tratada,batch_size=None,verbose=1)[0]
    return predicciones

def analizar_resultado(resultado, Labels):
    idx=np.argmax(resultado)
    label=Labels.classes_[idx]
    return label, idx

def mostrar_resultado(result, label, idx, imagen,fps):
    lab="{}: {:.2f}% FPS={:.2f}".format(label,result[idx]*100,fps)
    cv2.putText(imagen,lab,(10,25),cv2.FONT_HERSHEY_SIMPLEX,0.7,(0,255,0),2)
    cv2.imshow("resultado",imagen)
    print(lab)
    print("\n")

Modelo,Labels,Stream,rawCapture = Inicializacion("/home/pi/ModelsTesting")


for (i, frame) in enumerate(Stream):
        # grab the raw NumPy array representing the image, then initialize the timestamp
          # and occupied/unoccupied text
          image = frame.array
          start=time.process_time()
          result=evaluar_imagen(tratar_imagen(image),Modelo)
          label,idx=analizar_resultado(result,Labels)
          end=time.process_time()
          mostrar_resultado(result,label,idx,image,(1/((end-start))))
          key = cv2.waitKey(1) & 0xFF
          # clear the stream in preparation for the next frame
          rawCapture.truncate(0)
          #time.sleep(0.2)
          # if the `q` key was pressed, break from the loop
          if key == ord("q"):
                    break

cv2.destroyAllWindows()
parar_captura(Stream,rawCapture)

cv2.waitKey(0)
```