UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

# Aplicación de métodos de machine learning a la espectroscopía de protones acelerados por láser

Trabajo Fin de Grado

**Grado en Ingeniería Informática**


**Autor**: Javier Calatayud Giner

**Tutores**: María José Rodríguez Álvarez

Michael Seimetz


2018/2019

# AGRADECIMIENTOS

En primer lugar, me gustaría agradecer a mi tutor, Michael Seimetz, por haberme permitido trabajar con él en este proyecto, y su esfuerzo por guiarme e intentar ayudarme en todo momento.

Y, en segundo lugar, a mis padres, por su amor incondicional y total apoyo, tanto en los buenos y los malos momentos, sin los cuales no podría estar hoy aquí redactando este documento.

# Resumen

En este trabajo de fin de grado se ha desarrollado un sistema de clasificación automática de imágenes microscópicas basado en tecnología de Deep Learning. Estas imágenes son el resultado de un experimento en el que se hace colisionar protones acelerados contra un material detector, y consisten en un fondo gris con trazas con forma circular. La mayoría de las imágenes que se obtienen con este experimento no sirven debido a distintos factores como la falta de nitidez, por lo que es necesario clasificarlas. A su vez se ha implementado un sistema que extrae el radio y posición de las trazas usando técnicas de visión artificial.

**Palabras clave:** Inteligencia Artificial, Machine Learning, Deep Learning, Redes neuronales convolucionales, Keras, Vision Artificial, Transformada de Hough, Protones, Laser.

# Abstract

In this final project, a classification system for microscopic images based on Deep Learning technology has been developed. These images are the result of an experiment in which accelerated protons collide against a detector material. Most of the images obtained with this experiment cannot be used due to different factors such as lack of sharpness, so it is necessary to discard them. Also, I have implemented an algorithm that is able to extract automatically the radius of the traces contained in the images using artificial vision techniques.

**Keywords:** Artificial Intelligence, Machine Learning, Deep Learning, Convolutional neural networks, Keras, Computer Vision, Hough's transform, Laser, Protons.

# Tabla de contenidos

1. Introducción .......... 10
1.1 Motivación .......... 10
1.2 Problemas .......... 11
1.3 Objetivos .......... 12
2. Respuesta de CR39 a protones monoenergéticos .......... 13
3. Imágenes .......... 15
4. Herramientas .......... 20
Python .......... 20
ANACONDA .......... 20
Tensor Flow, Scikit Learn y Keras .......... 21
OpenCV .......... 21
PILLOW .......... 21
Numpy .......... 22
Matplotlib .......... 22
System y OS .......... 22
5. Deep Learning .......... 23
5.1 Inteligencia Artificial .......... 23
5.2 Redes Neuronales .......... 23
5.3 Problemas .......... 24
5.4 Capas Neuronales .......... 25
5.5 Funciones de Activación .......... 25
5.6 Redes convolucionales .......... 27
5.7 Subsampling .......... 28
5.8 Flattening .......... 28
5.9 Dropout .......... 29
5.10 Precisión del modelo .......... 29
5.11 Optimizadores .......... 30
5.12 Entrenamiento de la Red Neuronal .......... 30
5.13 Datos de Entrenamiento .......... 31
5.14 Creación de la red .......... 33
5.15 Resultados .......... 33
5.16 Aplicación .......... 35
6. Computer Vision .......... 36
6.1 Transformada de Hough .......... 36
6.2 Hough Circles .......... 37
6.3 Código .......... 38

6.4 Resultados ........ 39
6.5 Aplicación ........ 40
7. Implementación Final ........ 43
7.1 Resultados ........ 44
8. Conclusiones ........ 47
8.1 Valoración Personal ........ 47
8.2 Futuras Mejoras ........ 47
Referencias ........ 49
A. Implementación y Funcionamiento del Código ........ 51

# ÍNDICE DE FIGURAS

Figura 1: Ejemplos de muestras obtenidas durante el experimento 11

Figura 2: Ejemplo del output esperado por el programa 14

Figura 3: Muestra con trazas pequeñas//Muestra con trazas pequeñas 16

Figura 4: Muestra errónea debido a la aparición de la marca del láser 17

Figura 5: Muestras errónea y correcta con acumulación de círculos 17

Figura 6: Muestra errónea parcialmente oscura 18

Figura 7: Muestra errónea debido a la falta de trazas 18

Figura 8: Muestra con falta de nitidez 19

Figura 9: Representación de una neurona artificial 24

Figura 10: Representación de la función ReLU 25

Figura 11: Ejemplo input/output de la función ReLU 26

Figura 12: Formula de la función softmax 26

Figura 13: Perceptrón multicapa 27

Figura 14: Funcionamiento de la convolución 27

Figura 15: Operación de Max pooling 28

Figura 16: Operación de Flattening 29

Figura 17: Métricas del entrenamiento de una red 33

Figura 18: Evolución de la precisión y el valor de perdida de nuestra red 34

Figura 19: Diagrama de la arquitectura de nuestra red 34

Figura 20: Ejemplo de uso de la Transformada de Hough 36

Figura 21: Ejemplos de la aplicación de la transformada de Hough 40

Figura 22: Modelo de output en fichero de texto 44

Figura 23: Histograma sobre la distribución de los radios de las trazas 44

Figura 24: Histograma sobre los radios con una distribución inusual 45

Figura 25: Diagrama de puntos sobre la relación del radio con la energía 46

# 1. Introducción

Vivimos en la era de la información. El avance de las tecnologías ha propiciado que se generen miles de petabytes de información cada segundo gracias a los sistemas informáticos. La gran velocidad con la que actúan este tipo de sistemas ha provocado que su mayor ralentización la sufran cuando entra en juego la acción humana, surgiendo así una necesidad de automatización dentro de la sociedad entrando en juego así la inteligencia artificial.

La inteligencia artificial está a la orden del día [1]. Es habitual toparse a diario con diversos artículos desgranando otra de sus asombrosas aplicaciones, las cuales hace años habrían sido impensables y más propias de una película de ciencia ficción. Desde los coches autónomos hasta el reconocimiento de objetos en tiempo real, la IA ha dejado patente su gran potencia y transversalidad. Debido a esta última característica, es posible encontrar esta tecnología en cualquier área de aprendizaje que requiera automatización o que sea de utilidad poder predecir futuros eventos.

## 1.1 Motivación

Esta premisa nos motivó a aplicar este tipo de tecnología al campo de la física y, más concretamente, a la clasificación de imágenes microscópicas. Estas imágenes (Figura 1) son el resultado de un complejo experimento [2], en el que se fuerza la colisión de protones, acelerados mediante un campo electroestático, con una placa sensible a este tipo de partículas. Sobre ese chip, los investigadores realizan un proceso de revelado y un posterior fotografiado, utilizando un microscopio.

Las muestras consisten en una serie de circunferencias, las cuales llamaremos trazas, de color blanco o negro, dependiendo de las condiciones del experimento, sobre un fondo gris. El radio de estas trazas también variará, siendo las trazas más pequeñas de unos 5 µm, y llegando el radio de las más grandes a 40 µm.

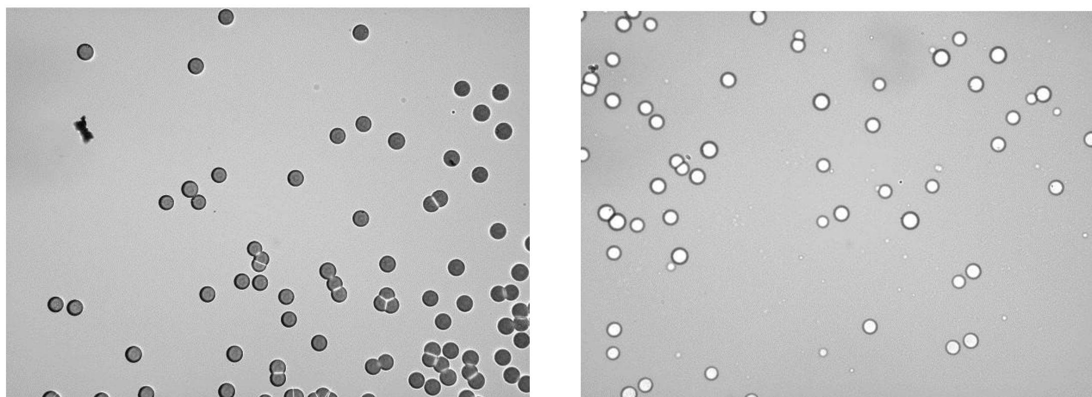


Figura 1: Ejemplos de muestras obtenidas mediante el experimento

Como resultado del fotografiado se obtendrán una gran cantidad de imágenes (entre 440 y 560), de las cuales, la mayoría, no serán aptas para el estudio. Debido a esto, es necesario realizar una clasificación del total de las imágenes, para determinar si son útiles o no. El problema que presenta dicha clasificación es que esta se realiza manualmente, por lo que consume una gran cantidad de tiempo, cosa que queremos solventar aplicando métodos de Deep Learning, de forma que sea una red neuronal convolucional la que se encargue de clasificar dichas imágenes, ahorrando una gran cantidad de tiempo y automatizando todo el proceso.

Una vez clasificadas las muestras, tenemos que realizar un proceso en el identificamos las trazas y extraemos su radio haciendo uso de la transformada Hough. Es importante extraer el radio de forma precisa ya que nos servirá para poder realizar una representación gráfica de la distribución.

## 1.2 Problemas

Ya hemos visto dos ejemplos de muestras anteriormente. Estas no siempre serán iguales, de hecho, factores como la distribución, el radio, el número y el color de las trazas, variarán de todas las formas posibles.

Esto será un gran inconveniente a la hora de entrenar una red neuronal convolucional, ya que será muy difícil para la red extraer cuales son los aspectos o patrones que validan una imagen. A su vez el número de muestras que tenemos para realizar dicho entrenamiento no es muy alto lo cual dificulta dicho aprendizaje.

Mis conocimientos en esta área también ralentizarán el desarrollo, ya que, al no pertenecer yo a la rama de Computación, todos los conocimientos que necesite los voy a adquirir de forma autodidacta.

## 1.3 Objetivos

Los objetivos pasaran por desarrollar un programa que sea capaz de clasificar las muestras obtenidas durante el experimento y extraer las trazas de estas, todo de forma automática. Dichas tareas las debe realizar en un lapso lo más acotado posible y de forma correcta.

El objetivo final de todo el proyecto será utilizar los radios obtenidos durante la extracción de trazas para crear una escala de calibración, la cual nos permita identificar la energía con la que se acelerado un protón solo sabiendo cual es el tamaño de la traza que ha dejado sobre el chip.

Para cumplir los objetivos vamos a utilizar una red neuronal convolucional, la cual construiremos y entrenaremos nosotros, y métodos de visión artificial para extraer la información sobre las trazas de las muestras.

Para solucionar los problemas relacionados con los conocimientos sobre redes neuronales, vamos a realizar un estudio en profundidad sobre la inteligencia artificial, haciendo hincapié en las redes neuronales. Los problemas que nos puedan dar la morfología de las trazas, los solucionaremos construyendo un software de apoyo que, haciendo uso de técnicas de visión por computador, extraiga manualmente determinadas características de las imágenes para determinar si estas son válidas. También tendremos como objetivo que la clasificación dependa lo menos posible de este software de apoyo.

# 2. Respuesta de CR39 a protones monoenergéticos

En este apartado explicaremos como se desarrolla el experimento de aceleración de protones, del que se obtienen las imágenes con las que vamos a trabajar.

El experimento [2] consiste en forzar la colisión de protones con una placa de un material detector, llamado CR-39, de un tamaño aproximado de 1 $cm^2$.

El material CR-39 es un polímero plástico que actuará como detector pasivo. La principal característica, y lo que hizo que se escogiese este material sobre otros, es que solo es sensible a partículas de alto LET (Linear Energy Transfer), ignorando, por tanto, los electrones y fotones.

Para que los protones colisionen es necesario acelerarlos en un campo eléctrico. La elección de este método de aceleración se debe a que los protones se cargarán con la misma energía, y por lo tanto las trazas serán de un radio similar. Esto nos servirá para relacionar el radio medio de las trazas con la energía de estos, creando así una escala de calibración, y que, cuando se utilicé otro método de aceleración en el que los protones no vayan cargados con la misma energía, sea posible identificar la energía de estos solo sabiendo el radio.

En caso de que se requiera que los protones tengan una energía inferior a 0,7 MeV, habrá que seguir un procedimiento un poco distinto, ya que el acelerador que utilizamos no nos permite acelerar por debajo de ese valor. Para llevarlo a cabo se colocará una placa de aluminio de unos 10μm de grosor. Esta placa retendrá parte de la energía de los protones, y por tanto se podrán tomar muestras por debajo de los 0,7 MeV. El inconveniente que tiene es que los tamaños de las trazas variaran más de lo normal, lo cual aumentará el error de las medidas que hagamos.

Al finalizar el proceso de aceleración de protones, se tiene que revelar el chip, haciendo uso de sosa caustica a 90 grados durante 4 horas; y realizar fotografías de este, donde se obtendrán unas 140 muestras por cada serie de fotografías. Cada una de estas series fotografía solo el 40% de la superficie del chip y se varia la distancia focal entre series de un mismo chip para obtener para obtener fotos nítidas de distintas zonas. De estas imágenes la mayoría no son aptas para el análisis (solo el 5% son aptas), ya sea por la gran acumulación de trazas, la mal calibración del microscopio o la presencia de la marca del láser, entre otros factores, lo cual hace necesario que estas sean clasificadas para evitar errores durante la extracción de información.

La parte final del experimento consiste en extraer la información. Para realizar esta tarea, ya existe un programa diseñado por mi tutor, Michael Seimetz, pero aun así nosotros realizaremos nuestra propia versión, para mejorar su eficiencia y poder encadenarlo con el sistema de clasificación, haciendo uso de las mismas técnicas.

Esta información es, principalmente, la cantidad de trazas que contiene, la posición de estas y su radio, con la que podremos calcular otras métricas como el radio medio o el error. Para extraer la información se utiliza la operación matemática conocida como transformada de Hough aplicada a circunferencias, la cual explicaremos más adelante, en la sección “Computer Vision”, la cual aprovecharemos para marcar las trazas sobre la imagen original, como se puede ver en la Figura 2.

Finalmente, cuando se han obtenido ya todos los datos, podemos realizar una representación gráfica de los mismos. Esta representación es en forma de histograma, ya que es la forma más fácil de ver la distribución de los radios, como se puede comprobar en la Figura 2. También hemos añadido el radio medio y el error en pixeles de imagen en la parte superior para tener toda la información extraída en un solo archivo.

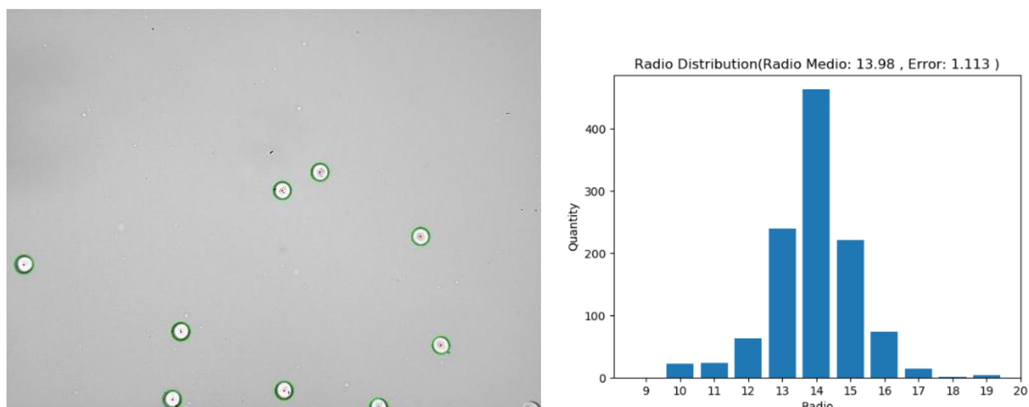



Figura 2: (Izquierda) Imagen en la que el programa ha marcado las imágenes detectadas. (Derecha)Ditribucion de los radios tras analizar una set de muestras. Ambas dos imágenes representarán el output esperado del programa.

# 3. Imágenes

En este apartado nos centraremos en exponer como serán las imágenes con las que trataremos. Las imágenes son el resultado del experimento anterior. Estas fotografías son extraídas de una placa contra la que han colisionado protones, los cuales han dejado trazas de forma circular.

Las muestras varían enormemente en función de las condiciones del experimento, y las variaciones tienen que ver con el tamaño, color y distribución de las muestras, además, independientemente de las condiciones, las muestras pueden contener elementos perturbadores, que hagan inútil una imagen.

Al tratarse trazas circulares, el tamaño lo podremos medir usando su radio, y para cuantificarlo, usaremos los pixeles de imagen como medida, los cuales pueden ser convertidos fácilmente a micrómetros. Como hemos sugerido en el párrafo anterior, esta métrica varía mucho, encontrándonos las trazas más pequeñas con un radio medio de apenas 10 pixeles y las más grandes con un tamaño de 50 pixeles (Figura 3). La información del radio medio será la medida más interesante por lo que necesitaremos precisión a la hora de extraerla y para ello utilizaremos la transformada de Hough, una técnica matemática para identificar formas en imágenes, la cual explicaremos en detalle en la sección de 'Computer Vision'. El tamaño de las trazas es generalmente homogéneo, ya que al haber sido cargadas con la misma energía todas las partículas, la traza deberá ser igual, y, en caso de que el tamaño varíe ligeramente, lo achacaremos a una medición imprecisa. Esto no siempre será así ya que, cuando se intente cargar a los protones por debajo de 0,7 MeV, la distribución se distorsionará, y los radios variaran mucho más.

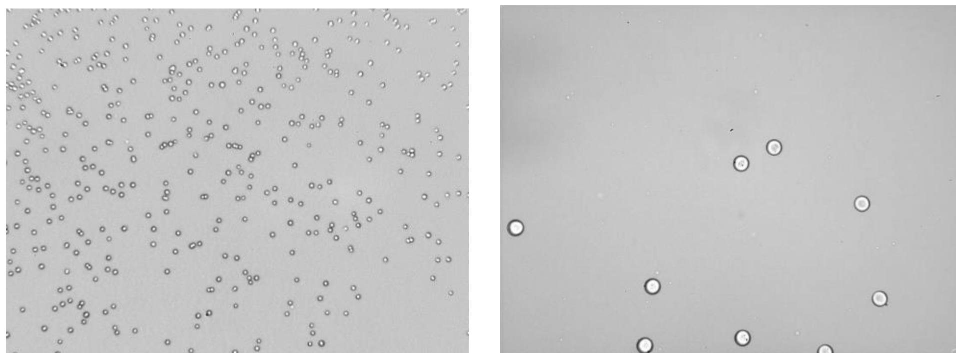


Figura 3: (Izquierda) Ejemplo de una muestra con las trazas pequeñas. (Derecha) Ejemplo de muestra con trazas grandes

Las trazas son siempre o blancas o negras, aunque no tienen por qué ser todas las trazas de una muestra del mismo color. Lo primero que realizará nuestro software al iniciarse, será separar las muestras según esta característica. Como el color de las trazas, generalmente, es heterogéneo, consideraremos que una muestra pertenece a un grupo si la mayoría de las circunferencias son de un mismo color. Durante el procesado, se les dará más valor a las muestras de color blanco, analizando estas primero. Las muestras negras solo se analizarán cuando no haya muestras blancas correctas.

Por último, la distribución de las trazas es aleatoria, pudiendo encontrar muestras con cualquier cantidad de trazas en cualquier posición de la imagen, por lo que será imposible saber la posición de estas antes de analizarlas.

Las muestras están agrupadas en 'ráfagas' de fotografías, en las cuales se generarán unas 140 fotografías, en cada una. Sobre cada placa, o 'chip', se realizarán 3 o 4 ráfagas, por tanto, se dispondrán de 440 a 560 imágenes por cada uno. Estos chips se agrupan en series, constando cada una de doce. En cada serie se han utilizado unas condiciones distintas para llevar a cabo el experimento, y con cada chip se habrá utilizado un nivel distinto de energía para cargar las partículas. Esta información será la que nos permita posteriormente relacionar el tamaño medio de las trazas de un chip con la energía empleada.

De todas las imágenes que tomemos, no todas serán aptas para el estudio, de hecho, solo el 5% de las imágenes se podrán utilizar para extraer la información de ellas, debido a distinto factores que invalidan al resto.

Los principales motivos para descartar una imagen serán los siguientes:

Aparece la marca del láser: El fabricante del chip detector graba con láser el número de serie y realiza una marca a la izquierda de él. Al realizar el fotografiado es posible que aparezca dicha marca en las imágenes como se puede observar en la Figura 4. En cada ráfaga de imágenes hay unas 15 imágenes con este tipo de anomalía, y aparecerán siempre al principio. Puede ocurrir que aparezca esta marca pero que la imagen aun pueda servir para el análisis, pero es tan raro este caso que siempre las descartaremos.

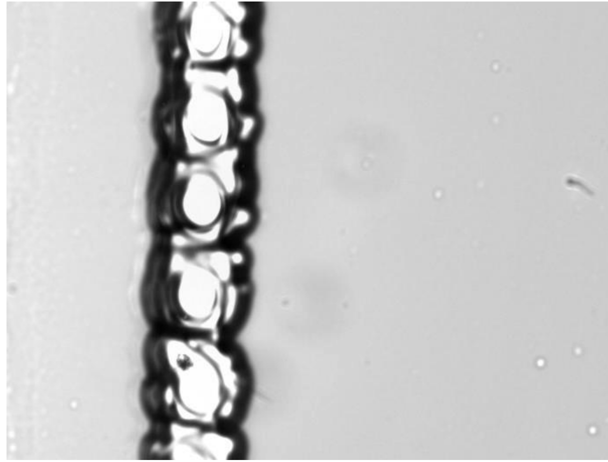


Figura 4: Ejemplo de imagen errónea debido a la aparición de la marca del laser

Gran acumulación de círculos: La acumulación de las circunferencias también es un problema para la extracción de información. Esta acumulación podrá aparecer de dos formas como se ven en las dos imágenes inferiores. En el primer subtipo de estas, la acumulación es tal que no se distinguirán las trazas, distorsionando la morfología de estos y recordando a la estática de un televisor (Figura 5). El segundo tipo de estas habrá que tratarlo con cuidado, ya que, a pesar de existir la acumulación, puede que se distingan las trazas, y por tanto la imagen sea útil.

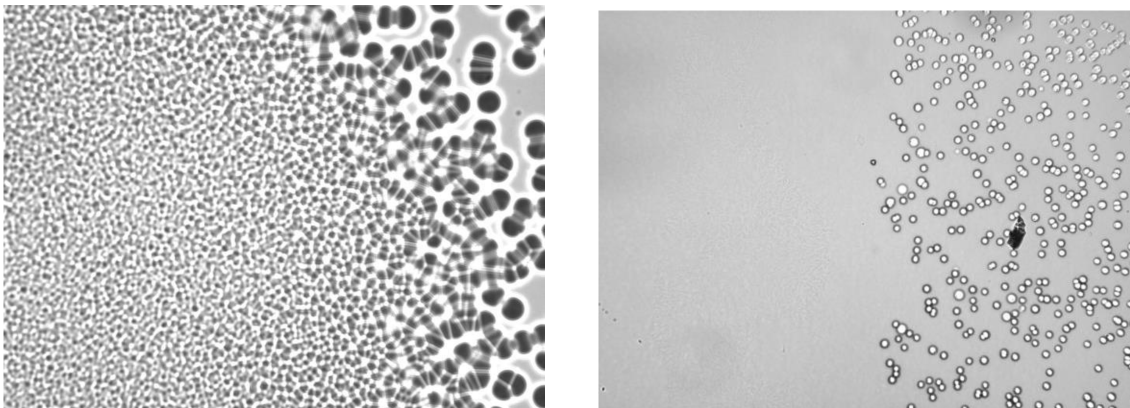


Figura 5: Ejemplo de imagen errónea debido a la acumulación de círculos(Izquierda) e imagen correcta con acumulación de círculos(Derecha

Parcialmente oscuras: Debido a como se toman las fotografías, también puede ocurrir que estas sean parcial o completamente oscuras (Figura 6).

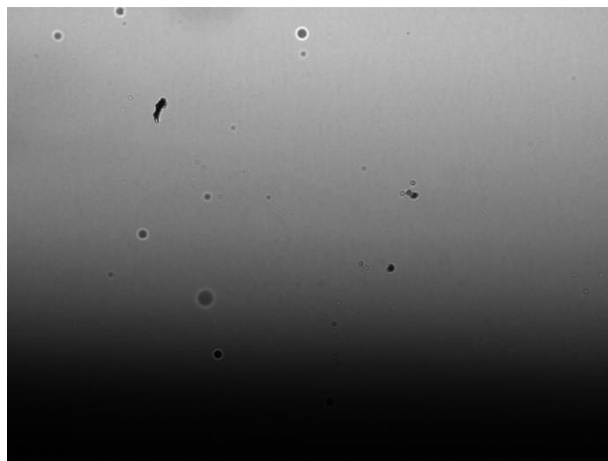


Figura 6: Ejemplo de imagen errónea debido a ser oscura

Imágenes vacías: Las imágenes vacías (Figura 7) también serán comunes entre las muestras. Para detectar este tipo de imágenes estableceremos un mínimo de círculos, y en caso de que una imagen no lo supere se considerará vacía. Este mínimo será de unos 5 círculos, y serán más comunes en trazas de radios grandes, ya que el número de trazas en el chip será menor.

Figura 7: Ejemplo de imagen errónea debido a la falta de muestras

Imagen no nítida: Este será el tipo de imagen más difícil de descartar ya que su error no será tan obvio como en el resto. La clave para que estas imágenes no sean nítidas reside en el tamaño del borde en la mayoría de los casos (Figura 8), y por tanto serán difíciles de descartar con una red neuronal por lo que crearemos un software de apoyo para analizar el tamaño de los bordes. El descartar o no alguna de estas imágenes dependerá muchas veces del criterio subjetivo del observador por lo que ambas opciones serán válidas.

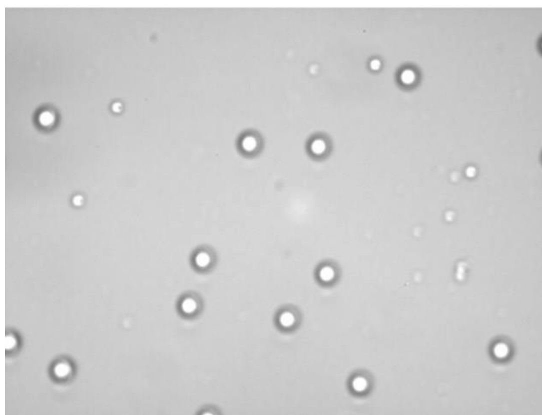


Figura 8: Ejemplo de imagen errónea debido a la falta de nitidez

Por último, cabe comentar que las imágenes las recibiremos con una resolución nativa de 2k(2048x1536), lo que provocará que tengamos que redimensionarlas obligatoriamente, ya que, de no hacerlo, será imposible entrenar una red neuronal con los recursos informáticos de los que disponemos, y a su vez el redimensionamiento acarreara más problemas.

Las imágenes más difíciles de clasificar son aquellas en las que los detalles que podrían invalidarlas son más discretos, como hemos comentado en las imágenes que no son nítidas. Este borde es de unos pocos pixeles por lo tanto si efectuamos el redimensionamiento, toda esta información se perderá o quedará muy reducida, siendo casi imposible detectar por nuestra red.

Para solucionar este problema, enfocaremos la red neuronal en clasificar las tres primeras clases de imágenes, aquellas que tienen la marca del láser, las que son parcialmente oscuras y en las que existe acumulación de círculos; y con el software de apoyo analizaremos el resto, de forma que, si no encuentra trazas, se considere que es una imagen vacía, y en caso de encontrar círculos, poder extraer los detalles que se pierden en el redimensionado y clasificar la imagen.

# 4. Herramientas

En esta parte hablaremos de la tecnología usada durante el proyecto. Por tanto, incluiremos el lenguaje de programación escogido, el entorno de desarrollo y las librerías, haciendo hincapié en la librería que hemos utilizado para crear la red neuronal, debido a que existen diversas alternativas igualmente válidas.

## Python

La elección del lenguaje de programación a utilizar es una decisión muy importante, no solo porque el hecho de estar familiarizado con el mismo va a facilitar el desarrollo, sino también porque de esta elección van a depender las herramientas de las que dispongas.

El lenguaje de programación que he elegido es Python [3], por su gran versatilidad y mi familiaridad con su sintaxis.

Python es un lenguaje de alto nivel multiparadigma creado a finales de la década de los 80. Su principal característica es que deshecha ideas implementadas en otros lenguajes como los ';' al final de las líneas y las llaves ('{}'), los cuales sustituye por un sistema de tabulaciones. Además, usa un sistema de tipado dinámico de variables, permitiéndote asignar cualquier valor a cualquier variable sin necesidad de declararla. Python ha sufrido un gran auge desde 2015 hasta convertirse en uno de los más utilizado, de forma simultánea al crecimiento de la inteligencia artificial. Estos dos hechos están relacionados ya que Python posee una gran cantidad de librerías que permiten el desarrollo de este tipo de tecnología, siendo ese uno de los motivos por los que escogí este lenguaje. Como inconveniente tiene que no es un lenguaje que se caracterice por ser el más eficiente, siendo superado por otros lenguajes como C++, aunque esto se soluciona, en parte, con la importación de módulos, la cual te permite que estos estén escritos en un lenguaje distinto a Python.

## ANACONDA

Para desarrollar en Python he elegido la plataforma Anaconda [4]. Anaconda es una distribución de Python muy utilizada en el campo del Machine Learning. Su principal ventaja es que te permite generar espacios de trabajo de forma rápida e intuitiva, con aquellas librerías que necesites, resolviendo de forma automática todas las dependías entre estas.

No conozco una alternativa a este software así que cuando no disponíamos de él, utilizábamos una maquina Ubuntu en la que instalamos todas las librerías, y ejecutábamos el programa desde la terminal.

## Tensor Flow, Scikit Learn y Keras

Otra decisión importante durante el desarrollo fue la librería de inteligencia artificial que íbamos a utilizar. Para tomar esta decisión barajamos tres opciones: Tensor Flow, Sklearn y Keras.

La primera, Tensor Flow [5], es una librería desarrollada por Google y presentada en 2015. Esta escrita en C++ y CUDA, y fue la segunda opción dentro de estas tres. Su principal ventaja respecto a las otras dos fue el grado de personalización que te permite. Esta característica provoca que esta librería no sea amigable, ya que será el propio usuario quien tenga que diseñar los componentes que quiera en su red.

Sklearn (Scikit Learn) [6] es otra librería desarrollada por Google en 2010. Su mayor fortaleza respecto al resto es su gran sencillez. Esta librería permite construir una red con una sola línea de código, lo que impedirá una gran personalización. Además, entre su repertorio no hay redes neuronales convolucionales, lo que la descarto definitivamente.

La última, y nuestra elección, es Keras [7]. Keras es una librería de código abierto lanzada en 2015. Esta se trata de una implementación de alto nivel de Tensor Flow, la cual está diseñada para ser fácil de usar, modular y compatible con esta. La librería ofrece muchas herramientas como capas y funciones de activación ya implementadas, por lo que permite al usuario centrarse en el diseño de la arquitectura de la red.
Además, entre su repertorio hay capas convolucionales, lo que posibilita crear una red de este tipo. Keras también ofrece arquitecturas de redes "famosas" que tendremos disponibles en caso de querer probarlas.

Existen otras librerías que, se también barajamos entre otras opciones, pero que descartamos por razones distintas como Pytorch, Theano o DeepLearning4J.

## OpenCV

Opencv [8] es una librería de visión por computador muy extendida. Apareció por primera vez en 1999 y sus utilidades giran en torno al reconocimiento de formas, objetos, calibración de cámaras y visión robótica. La función que más emplearemos de esta librería es 'HoughCircles'. Dicha función nos permitirá realizar la transformada de Hough aplicada a circunferencias sobre una imagen, tomando unos pocos parámetros como el radio mínimo y máximo, o la separación entre los centros de la circunferencia. A parte la librería también nos permitirá dibujar esos círculos sobre la imagen, y realizar otras modificaciones, como escalarla o pasarla a una escala de grises.

## PILLOW

PILLOW [9] (también conocida como PIL) es una librería de procesamiento de imágenes. La principal utilidad que nos brindará es la extracción de pixeles de una imagen como si se tratase de un matriz. A parte utilizará el mismo sistema de numeración que Opencv por lo que se complementaran muy bien. Esta librería nos servirá para determinar el color de las trazas y el tamaño del borde de las trazas.

## Numpy

Numpy [10] es una librería que da acceso a estructuras como vectores y matrices, además de proveer operaciones matemáticas de alto nivel. Esta librería tendrá su importancia dentro del proyecto ya que algunas de las funciones que utilicemos necesitaran que la entrada sea una de las estructuras de las que nos provee.

## Matplotlib

Matplotlib [11] es una librería que nos permitirá generar con unas pocas líneas de código todas las representaciones graficas que necesitemos. Además, nos provee de herramientas propias de otros lenguajes como Matlab, que nos facilita trabajar con matrices.

## System y OS

Tanto system como OS son librerías nativas de Python y nos darán herramientas para interactuar con el sistema operativo. Principalmente utilizaremos estas herramientas para crear, modificar y borrar archivos y directorios.

Estas no han sido las únicas herramientas que vamos a utilizar, pero si las de más importancia. Entre las que no hemos explicado se encontrarían aquellas las librerías que realizan las tareas menos importantes o herramientas más secundarias como Google Drive para almacenar en la nube ciertos documentos.

# 5. Deep Learning

En este apartado, voy a hacer una introducción a las redes neuronales, centrándome en sus distintos componentes, y a explicar cómo hemos implementado la que utilizamos en nuestro software.

## 5.1 Inteligencia Artificial

La inteligencia artificial es el campo de estudio que busca dotar a las máquinas de habilidades asociadas a la inteligencia humana, principalmente la habilidad de realizar tareas cognitivas en las cuales reconocen las características de su entorno y reaccionan a ellas, para llevar a cabo con éxito una tarea, y engloba desde la planificación automática de eventos hasta el reconocimiento de imágenes y audio.

Dentro de estos conceptos encontramos muchos subcampos, entre los que nos interesa destacar el Machine Learning o aprendizaje automático. La principal característica de esta área es que el software desarrollado 'aprende', y mejora su eficiencia a medida que se ejecuta. Esto abrió en su momento un océano de posibilidades ya que permitía realizar tareas complejas sin necesidad de que estas estuviesen entre sus rasgos de nacimiento. A partir del aprendizaje automático surgió una rama la cual englobó los algoritmos más complejos de este campo, el Deep Learning.

El Deep Learning, o aprendizaje profundo es el área más avanzada de la inteligencia artificial, en la que se han desarrollado estructuras capaces de aprender a partir de estructuras complejas basadas en vectores y tratar con cantidades inmensas de datos, siendo las redes neuronales su mayor exponente.

## 5.2 Redes Neuronales

Una red neuronal [12] es un modelo matemático que pretende asemejarse a un cerebro humano. Este tipo de estructuras son diseñadas para ser 'entrenadas' y poder reconocer patrones numéricos en representaciones vectoriales y se caracterizan por estar compuestas por diversas capas conectadas. Cada capa recibe la información de la capa anterior, la procesa y la envía a la siguiente capa.

Las capas están formadas por neuronas computacionales [13]. Estas neuronas se tratan de otro modelo matemático, también conocido como perceptrón, el cual asigna un peso a cada uno de sus inputs, y calcula su output realizando una suma ponderada de los mismos, más un término independiente conocido como bias, separando así el espacio con una recta plano o hiperplano, dependiendo del número de inputs. Con una solo neurona se separará el espacio en dos regiones, las cuales se podrán utilizar para separar dos tipos de datos.

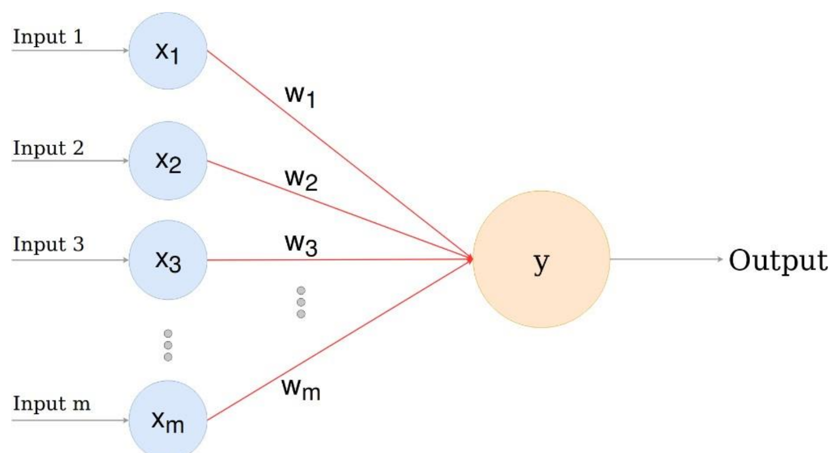



Figura 9: Representación de una neurona artificial

Para que este modelo funcione correctamente, se necesita un entrenamiento previo, el cual ajuste el valor del vector de pesos. Para realizar el entrenamiento es necesario tener valores de prueba ya clasificados, lo cuales consistan en un input y el output que se espera para dichos valores de entrada. El modelo recibe los valores de prueba y calcula el output. En caso de ser correcto, no hace nada, pero en caso de no serlo, ajusta los pesos resolviendo la fórmula que aparece a continuación.

$$w(j)' = w(j) + \alpha(\delta - y)x(j)$$

'w' representa el vector de pesos
'α' es la tasa de aprendizaje
'δ' es la salida esperada
'j' representa el elemento que estamos actualizando
'y' representa la salida que hemos obtenido
'x' es el vector de entrada

Como se puede comprobar contra mayor sea el error cometido, más variara el vector de pesos.

## 5.3 Problemas

Este modelo con una única neurona presento diversos problemas. El primer problema que surgió fue la incapacidad de separar el espacio en más de dos regiones, y, por tanto, el no poder tratar con más de dos clases de datos.

El segundo y más importante problema, fue la imposibilidad de unir varios de estos modelos secuencialmente. Esta es una restricción de los modelos de regresión lineal, los cuales al ser concatenados pueden simplificarse en uno solo.

## 5.4 Capas Neuronales

Las capas neuronales, son el resultado de la unión en paralelo de varias neuronas artificiales, y la solución al primer problema del apartado anterior. Estas estructuras permitieron realizar una clasificación más completa, y al unirlas secuencialmente, darían lugar a las redes neuronales actuales, cosa que sería imposible hasta resolver el segundo problema.

Las capas más extendidas son las conocidas como 'fully connected' y se caracterizan porque cada una de sus neuronas recibe el output de todas las neuronas de la capa anterior.

## 5.5 Funciones de Activación

Las funciones de activación [14] son unas funciones matemáticas que se ejecutan sobre la salida de las neuronas y buscan resolver el segundo problema. Estas funciones están diseñadas para modificar la salida de las neuronas de forma que sea posible concatenar varias de ellas, quitándoles linealidad. Estas funciones serán diversas y realizarán funciones como normalizar la salida o transformarla en una distribución de porcentajes.

### Función ReLU

La función ReLU o Rectified Lineal Unit se puede ver representada en la figura 11. La principal característica de esta función es que elimina los valores negativos convirtiéndolos en 0, dejando los valores positivos intactos. También es conocida por tener un coste muy bajo, pero puede causar problemas a la hora de actualizar los pesos de la neurona. Será especialmente útil cuando trabajemos con imágenes ya que las imágenes son no lineares por naturales, y esta función realzará esa característica. Su transformación en una imagen en blanco y negro como la nuestra será eliminar los valores negros, es decir negativos, dejando solo los tonos grises y blancos, como se ve en la figura 11.

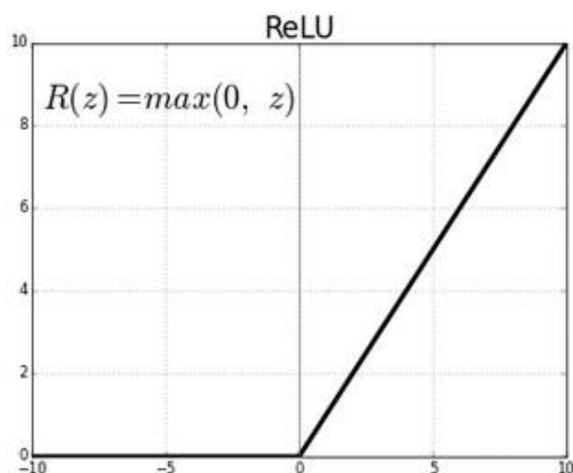



Figura 10: Representación gráfica de la función ReLU

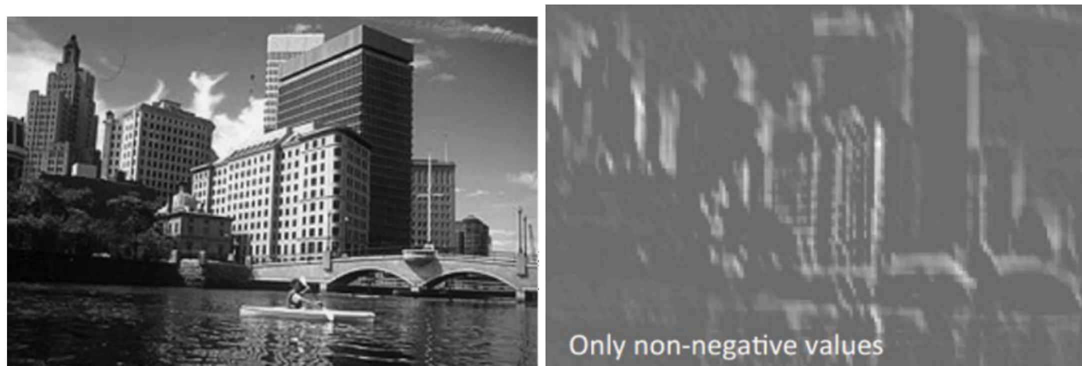



Figura 11: (Derecha)Input de la función ReLU. (Izquierda)Output de la función para dicho input

## Función Softmax

La función softmax estará asociada a la capa de salida de la red, y se encarga de transformar su entrada en una distribución de probabilidades con un porcentaje asociado a cada clase. Esta función será útil en problemas de clasificación y se definirá por la fórmula de la figura 12.

$$f(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Figura 12: Formula de la función softmax

Como es la más confusa de las funciones voy a poner un ejemplo de su uso. Pongamos que tenemos 4 clases y que cada clase tiene un numero de apariciones asociado, dando como resultado un vector así: [4, 1, 2, 3].

Sabiendo la cifra de apariciones, las utilizaríamos para exponenciar '*e*', obteniendo el resultado [54.6, 2.72, 7.39,20.09]; y así dividir cada uno de los resultados individuales entre el sumatorio de todos. El resultado final de aplicar la función sería [54.6, 2.72, 7.39,20.09], indicándonos así que, para la distribución inicial, lo más probable es que un supuesto input perteneciese a la clase 1 por tener el mayor porcentaje asociado.

Estas funciones darán lugar a el modelo del perceptrón multicapa (Figura 13), la arquitectura de red neuronal clásica, en la que se introducen datos por la capa de entrada, se procesa la información en cada capa y se trasmite, dando lugar a un output en la capa de salida.

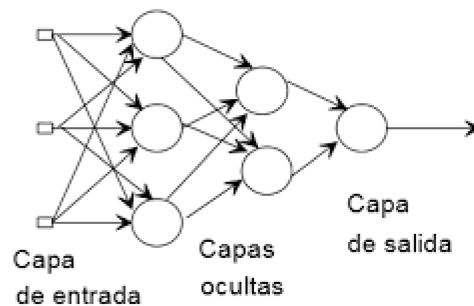



Figura 13: Arquitectura del perceptrón multicapa

## 5.6 Redes convolucionales

Las redes neuronales convolucionales [15] son un tipo de arquitectura especialmente diseñada para tratar con representaciones de datos matriciales, las cuales son muy importantes dentro del campo de la visión por computador. Estas redes obtendrán su nombre de un tipo especial de capa que contienen.

Las capas convolucionales realizarán lo que se conoce como convolución sobre su input. Una convolución es una operación la cual extrae las características más importantes de una imagen, utilizando una matriz conocida como filtro, realizando con ella una serie de operaciones sobre la imagen original. Para llevar a cabo su tarea, la red dará más valor a las características más importantes de la imagen.

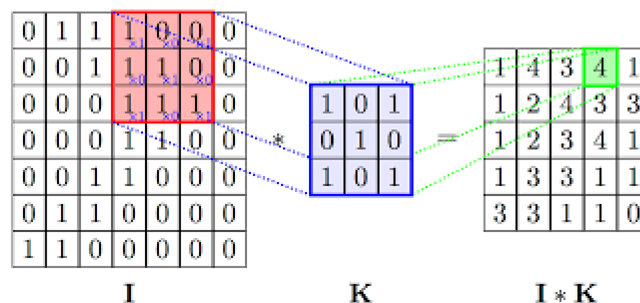



Figura 14: representación del funcionamiento de una convolución

Dentro de las redes neuronales también existirán otros tipos de operaciones que se podrán realizar.

## 5.7 Subsampling

El subsampling [16] consiste en aplicar un filtro de salida a las capas convolucionales para que elimine aquellas partes que tienen menos importancia, reduciendo el número de neuronas necesarias en la siguiente capa, y por tanto mejorando el rendimiento.
Estas operaciones acompañaran a una capa convolucional, ya que mediante las convoluciones se les dará más valor a las características más importantes de las imágenes, y el subsampling eliminará el resto. Su tarea la realiza dividiendo la matriz en matrices cuadradas del mismo tamaño y escogiendo el valor más alto en cada una de ella, eliminando el resto.

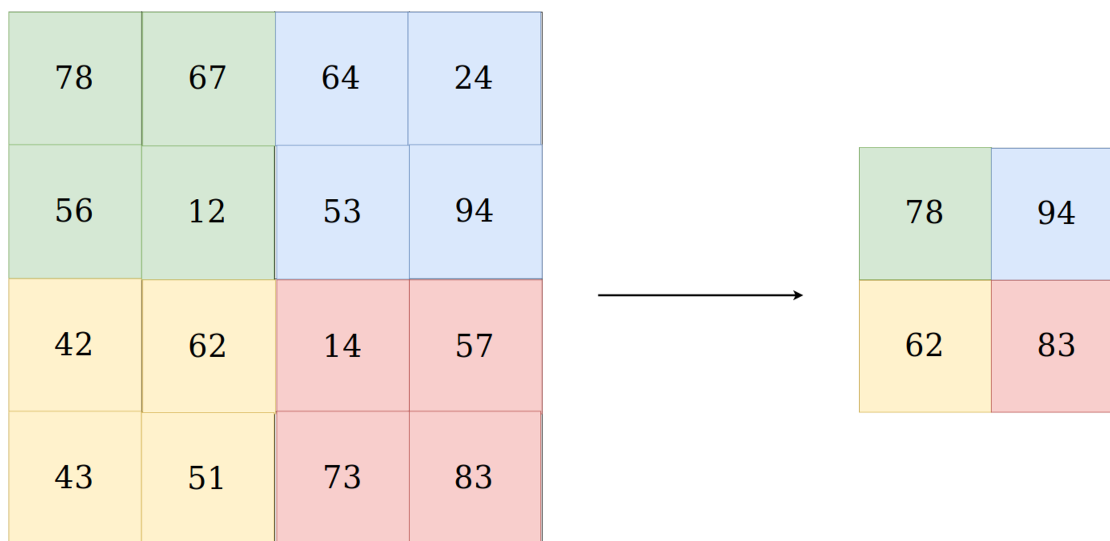



Figura 15: Ejemplo del funcionamiento de la operación max pooling

La implementación en Keras de esta operación se hará añadiendo una capa llamada max pooling y nos pedirá por parámetros el tamaño de las submatrices cuadradas, que en nuestra implementación serán de un tamaño de 2x2.

## 5.8 Flattening

Las capas convolucionales y las arquitecturas clásicas no pueden ir conectadas de forma natural, ya que trabajan con estructuras de datos distintas (Las primeras con matrices y las segundas con vectores). Para resolver esta incompatibilidad, se tiene que hacer uso de una operación que convierta el primer tipo de datos en el segundo. Para ello se utiliza la operación Flattening [17].

Esta operación simplemente transforma una matriz (imagen 16) en un vector. Para ello coloca las filas de forma consecutiva.

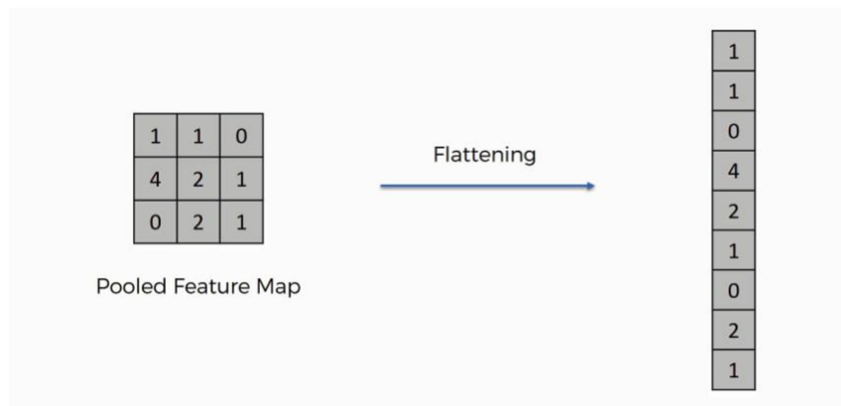



Figura 16: representación del funcionamiento de la operación flatten

En Keras, para realizar esta operación utilizaremos una capa Flatten, la cual colocaremos entre la última capa max pooling y la primera capa fully connected, y no recibirá argumentos.

## 5.9 Dropout

Esta función recibe un parámetro de probabilidad X, el cual representa la probabilidad de que uno de los nodos de la siguiente capa sea desactivado. Haciendo esto lo que se consigue es evitar que haya overfitting, forzando al resto de neuronas que no han sido desactivadas a trabajar más. El dropout solo lo usaremos durante el entrenamiento, aunque a veces puede ser ventajoso utilizarlo durante el testeo, y servirá para acabar con los problemas de overfitting que veremos más adelante.

## 5.10 Precisión del modelo

Para poder medir de forma numérica la precisión de nuestro modelo, utilizaremos la función de perdida [18]. Cada vez que le pasemos una nueva imagen a la red durante el entrenamiento, esta intentará clasificarla. En caso de que la consiga clasificar bien disminuirá este valor, y en el caso contrario, aumentará. Nuestro objetivo es que el valor de perdida sea lo más bajo posible lo que implicará que el modelo es más preciso.

La función que utilizaremos para calcular el valor de perdida se llama categorical crossentropy, la cual será útil cuando los valores de salida se mueven entre 0 y 1, e intentamos hacer una clasificación con más de dos categorías. Esta función se calculará con la siguiente formula:

$$CE = \sum_{C=1}^{M} y_c \log(p_c)$$

Donde $C$ es el número de clases, $y$ es el valor esperado y $p$ es el valor predicho.

## 5.11 Optimizadores

Para obtener el valor mínimo de perdida se suele utilizar el método del descenso del gradiente [19]. Este método calcula como afectarían pequeñas variaciones en el vector de pesos al valor de perdida calculando las derivadas parciales de los mismos. Los pesos se actualizarán en función al learning rate.

El learning rate será un valor predefinido por el programador, que indicará a la red cuanto se deberán actualizar los pesos en cada iteración del algoritmo del descenso del gradiente. Este no puede ser arbitrario ya que en caso de que sea muy pequeño, ralentizaría el entrenamiento; y en caso de que sea muy grande, puede causar problemas impidiendo que se alcance el valor óptimo. Para evitar este tipo de problemas existe un algoritmo llamado learning rate decay. Este algoritmo mantiene alta la tasa de aprendizaje al principio de la ejecución y la va disminuyendo a medida que avance la ejecución haciendo que al principio se ejecute rápido y que al final sí que se pueda llegar al valor óptimo de los vectores de pesos.

Keras nos dará también varias opciones entre los optimizadores, pero nosotros reduciremos las opciones a dos SGD y ADAM.

SGD (Stocastic Gradient Descent) es una variación del algoritmo del descenso del gradiente, cuya principal característica es que utiliza datos previamente entrenados para realizar sus cálculos, reduciendo así el coste computacional, a costa de necesitar más pasos para alcanzar el objetivo. Este optimizador no variará la tasa de aprendizaje durante su ejecución.

ADAM es otro algoritmo de optimización que partirá del anterior. La principal diferencia es que este utiliza el algoritmo Learning Rate Decay. También se caracteriza por usar el momentum. Este concepto se basa en actualizar el vector de pesos teniendo en cuenta la actualización anterior.

## 5.12 Entrenamiento de la Red Neuronal

Una vez diseñada la red neuronal, el siguiente paso será entrenarla. El entrenamiento se realiza utilizando un data set, el cual contendrá imágenes y una etiqueta indicando a que clase pertenece cada una de las imágenes. Durante el entrenamiento, la red intenta clasificar dichas imágenes y en caso de fallar, ajustará los vectores de pesos asociados a cada una de las neuronas.

El entrenamiento se divide en épocas. Durante cada época se iterará sobre el total de las imágenes, o un subconjunto de ellas, dividiendo cada época en iteraciones de tiempo más pequeñas conocidas como steps. En cada step se le pasará a la red un numero de imágenes igual a un parámetro conocido como batch size, y por lo tanto en cada época se utilizarán un numero de imágenes igual al batch size multiplicado por el número de pasos.

En nuestra red, utilizaremos el total de las imágenes en cada época, las cuales se dividirán en tanto steps como permita el batch size.

OVERFITTING

El overfitting [20] es un problema que puede surgir durante el entrenamiento como causa de que utilicemos las mismas imágenes para entrenar y testear nuestro modelo. Este fenómeno causará que la red aprenda a clasificar bien las muestras utilizadas pero que no sepa clasificar nuevas muestras que le pasemos, debido al reconocimiento de patrones incorrectos. Para evitar este problema, separaremos el 80% de las muestras para entrenar la red, y el 20% para testearla. Existen otras técnicas para evitar este fenómeno, como la terminación rápida, la cual interrumpe el entrenamiento cuando el valor de perdida deja de descender, o el dropout, que hemos visto anteriormente.

UNDERFITTING

Underfitting [20] será el efecto opuesto al overfitting. Este fenómeno se da cuando la red no ha sido entrenada los suficiente, lo que puede provocar que no haya aprendido patrones suficientes y por tanto que haga mal su trabajo. La solución a este problema pasa por entrenar más la red.

Para entrenar haremos uso de todos los conceptos vistos anteriormente, y añadiremos dos algoritmos que se encargaran de unir las piezas.

El algoritmo de forwardpropagation será el método habitual para usar la red neuronal. A la capa de entrada se le darán unos inputs, y esta lo trasmitirá a la siguiente capa, la cual los procesará e ira trasmitiendo sucesivamente, obteniendo unos outputs al final.

Una vez hemos ejecutado el algoritmo anterior, procederemos a utilizar el algoritmo backpropagation. Este se trata de la contraparte del anterior, y consiste en propagar desde la capa de salida el error de la red, de forma que cada neurona sepa cómo ha contribuido a este, y así poder modificar sus pesos de manera acorde. Este algoritmo fue fundamental para posibilitar las redes neuronales actuales, ya que de no existir las redes tendrían que ejecutar una estrategia de fuerza bruta, aumentando mucho el coste.

## 5.13 Datos de Entrenamiento

Los datos de entrenamiento que usaremos son imágenes con una gran resolución(2048x1536) en formato png las cuales deberán sufrir un procesamiento para que la red pueda trabajar con ellas. Lo primero que haremos será escalarlas a una resolución de 128x128 ya que, si intentamos crear el dataset con las imágenes originales, este proceso consumirá toda la memoria del ordenador hasta darnos un error por falta de memoria. También las convertiremos de formato RGB a una escala de grises para que la entrada de la red tenga solo 1 canal y no tres.

Una vez redimensionadas, las transformaremos en una matriz de la librería numpy, y normalizaremos los valores entre 0 y 1, diviendo el valor de cada píxel entre 255. A su vez, el total de las imágenes será dividido en dos grupos, uno que contendrá el 80% de las imágenes para el entrenamiento, y otro que contendrá el 20% restante, para el testeo. Los dos grupos de imágenes se le pasaran a la red. Al comenzar una nueva época, ajustará los pesos usando el grupo de entrenamiento, y luego pasará el segundo grupo para tener una idea de cómo es el rendimiento de la red.

Lo fundamental para que una red de este tipo pueda funcionar correctamente es poder disponer de datos para entrenarla. Dependiendo de la complejidad de la información estos datos van a cobrar más o menos importancia ya que va a ser más difícil para la red encontrar patrones en esta.

Una decisión importante es el número de clases en los que se podrá clasificar una imagen. Para ello vamos a agrupar las imágenes en distintas clases según sus características. Como hemos explicado en la sección de las imágenes, la red neuronal la utilizaremos para descartar 3 tipos de imágenes, aquellas en las que aparece la marca del láser, las que son parcialmente oscuras y aquellas en las que existe una acumulación de trazas. Por ello crearemos cuatro grupos de imágenes, los tres mencionados y uno de imágenes correctas.

El dataset que utilizaremos dispondrá de 550 imágenes correctas y 450 incorrectas para muestras blancas. Este dataset se puede ir ampliando a medida que se vayan realizando más veces el experimento, y así mejorar la precisión.

Para procesar los datos de entrenamiento y crear el dataset hemos utilizado un script llamado 'procesamiento.py'.

Para procesar las imágenes, primero recorrerá los directorios que le indiquemos, extrayendo las imágenes, y realizando el procesamiento que hemos en los párrafos anteriores. Una vez se realiza el procesamiento, las imágenes y la clase asociada a cada una las almacenamos en un array.

```
for dataset in data_dir_list:
    img_list = os.listdir("./" + dataset)

    for img in img_list:
        input_img = cv2.imread("./" + dataset + "/" + img)
        input_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
        input_img_resize = cv2.resize(input_img, (128,128))
        img_data_list.append(input_img_resize )
        labels_list.append(dataset)
```

Una vez hemos procesado y almacenado todas las imágenes, procesamos las imágenes como conjunto, normalizando los valores de los pixeles, expandiendo las dimensiones del conjunto para indicar al final que solo utilizaremos un canal y dividiendo el conjunto de las imágenes en los grupos de entrenamiento y testeo.

```
img_data = np.array(img_data_list)
img_data = img_data.astype('float')
#nomaliza los valores
img_data /= 255
labels = np.array(labels_list)
#convierte las categorias numericas en vectores; ejemplo 1-->[1,0,0,0], 2-->[0,1,0,0]
Y = np_utils.to_categorical(labels, 4)
#Expande las dimensiones para indicar el numero de canales
img_data = np.expand_dims(img_data, axis = 4)
#mezcla las iamgenes
x,y = shuffle(img_data,Y, random_state=2)
#crear los grupos de entrenamiento y testeo
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=2)
```

## 5.14 Creación de la red

La red que hemos construido constará de 8 capas. Las dos primeras capas son convolucionales, las cuales darán más valor a las características más importantes de las imágenes, con una ReLU asociada, la cual eliminará los valores negros. Estas dos capas se combinarán con una Max pooling, la cual reducirá el tamaño de la imagen, seguida de una operación DropOut.

Una vez realizado este proceso, a las imágenes les quedará pasar por un perceptrón multicapa compuesto por 2 capas fully-connected. Para enlazar la última capa max pooling y la primera fully-connected, haremos uso de una operación de flattening que convertirá la imagen de entrada en un vector. La siguiente capa también tendrá asociada una función ReLU, y le seguirá otro Dropout. Finalmente, la capa de salida tendrá asociada una función softmax, la cual nos dará un vector de probabilidades, con la probabilidad de que la imagen que hemos pasado pertenezca a la clase en cuestión.

## 5.15 Resultados

Una vez hemos construido la red, nos quedara entrenarla. Para ello llamaremos a la función 'fit' de la librería Keras, y empezará a mostrarnos métricas sobre cómo avanza el entrenamiento. Nuestra red está programada para ejecutar 20 épocas con un batch size de 32. Las métricas que se nos mostrarán en tiempo real serán las de la figura 17.

```
Train on 783 samples, validate on 196 samples
Epoch 1/20
783/783 [==============================] - 108s 137ms/step - loss: 2.9965 - acc: 0.4777
- val_loss: 0.9713 - val_acc: 0.6633
Epoch 2/20
783/783 [==============================] - 89s 113ms/step - loss: 0.8501 - acc: 0.6999
- val_loss: 0.7566 - val_acc: 0.8010
Epoch 3/20
783/783 [==============================] - 101s 129ms/step - loss: 0.5576 - acc: 0.7995
- val_loss: 0.5070 - val_acc: 0.8010
Epoch 4/20
783/783 [==============================] - 107s 136ms/step - loss: 0.3796 - acc: 0.8531
- val_loss: 0.4623 - val_acc: 0.8571
```

Figura 17: Métricas del entrenamiento de la red neuronal

Una vez ha terminado el entrenamiento podremos iniciar el framework TensorBoard. Este software nos proporcionara representaciones graficas sobre cómo ha ido el entrenamiento. En la figura 18, tenemos la evolución de la precisión, y en la figura, la evolución del valor de perdida.

El valor de precisión se situará al final en 0.99, lo que implica que al final del entrenamiento acertaba un 99% de las veces, y el valor de perdida en 0.03.

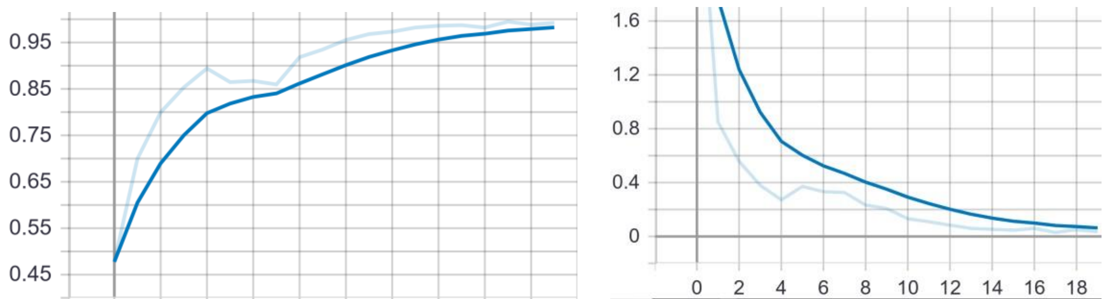



Figura 18: A la izquierda evolución del porcentaje de precisión de la red. A la derecha evolución del valor de perdida durante el entrenamiento.

Estos valores son lo suficientemente positivos como para asegurar que la red realizará su trabajo, pero sin olvidar que se dan porque las imágenes en las que la clasificación es más delicada, las clasificaremos mediante el software de apoyo.

Durante el desarrollo del proyecto, hemos probado otro tipo de redes famosas como la red Resnet de Google o la red VGG. Estas redes son redes formadas por decenas de capas, las cuales están diseñadas para tratar con data sets de imágenes muy amplios. Los resultados ofrecidos por estas son ligeramente mejores a los nuestros, con el inconveniente de que su entrenamiento es mucho más duradero, por lo que nos quedamos con nuestra red.

TensorBoard también nos ofrece un diagrama de flujo en el que podemos ver la arquitectura de nuestra red (Figura 19).

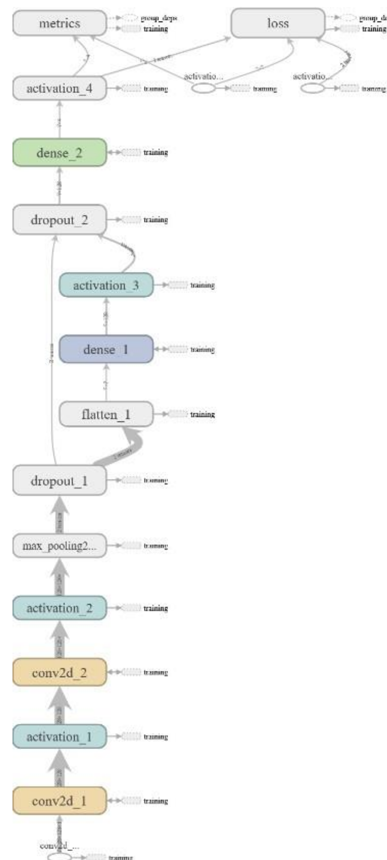



Figura 19: Arquitectura de la red neuronal

## 5.16 Aplicación

Para aplicar la red neuronal, primero tendremos que cargar el modelo de la red neuronal ya entrenada con la función “load_model”, entonces procesaremos las imágenes con el método prepareData, el cual nos devolverá un array con todas las imágenes procesadas y otro array con la ruta de dichas imágenes; y se las pasaremos a la función predict de la librería Keras. Como resultado obtendremos una matriz en la que cada fila representa la predicción para cada imagen.

```
currentModel = keras.models.load_model("redConvolucional.h5")
dataPrepared = prepareData()
data = dataPrepared[0]
fileNames = dataPrepared[1]
predictions = currentModel.predict(data)
processPredictions(predictions, fileNames)
```

Una vez tenemos las predicciones, llamamos a la función “processPredictions”. Esta función iterará sobre las predicciones hechas, y de que la red haya predicho que una imagen no es correcta la moveremos a la carpeta de imágenes incorrectas.

```
numberPredictions = len(predictions)
#Creamos un bucle que va desde 0 hasta el numero de predicciones - 1
for index in range(0, numberPredictions):
    currentPrediction = predictions[index]
    currentFile = fileNames[index]
    #Si el porcentaje de la clase correcta es menor a la suma
    #de los porcentajes de las clases incorrectas, descartamos la imagen
    if currentPrediction[0] < currentPrediction[1] + currentPrediction[2] + current
        #Con esta funcion movemos el archivo al directorio de imagenes incorrectas
        moverArchivo(currentFile, dirTarget, dirBad)
```

# 6. Computer Vision

La visión por computador [21] es una disciplina de la inteligencia artificial cuyo objetivo es dotar a las máquinas de la habilidad de percibir y procesar imágenes reales. Este se trata de un gran campo que abarca desde la detección de formas geométricas simples hasta distinguir rostros [22] con el refuerzo del machine learning.

De entre todas las cosas que nos ofrece este campo nosotros nos quedaremos con la Transformada de Hough aplicada a circunferencias [23].

## 6.1 Transformada de Hough

La transformada de Hough es una técnica matemática que permite detectar figuras en una imagen siempre que la forma de estas pueda ser expresada matemáticamente.

Figura 20: Ejemplo de uso de la transformada de Hough. Los círculos amarillos representan las circunferencias detectadas.

La forma más simple de esta operación se obtiene cuando la aplicamos a rectas. Una recta se puede expresar con las coordenadas cartesianas (m, n) siendo $y = m*x + n$, o con coordenadas polares $(\rho, \theta)$ con $\rho = x * \cos\theta + y * \sin\theta$, donde $\rho$ es la distancia entre la recta y el origen; y $\theta$ es el ángulo que forma la recta con el vector director que pasa por el origen de coordenadas.

La implementación de la operación consiste en crear una matriz acumuladora con todas las posibles combinaciones de ρ y θ, a partir de los bordes detectados en la imagen. Para detectar los bordes será necesario disponer de un algoritmo que realice dicha tarea.

Luego para cada píxel de la imagen, si forma parte del borde de alguna figura, se busca está probando todos los posibles ángulos. En caso de acierto se aumenta su valor en 1 en la matriz acumuladora.

Al final del proceso, se escogen los puntos con valores más altos en el acumulador y se devuelven, obteniendo así las rectas detectadas en forma polar.

Para aplicarlo a circunferencias, el procedimiento es muy similar solo que hay que tener en cuenta que la ecuación de las circunferencias es (x-a) ^2 + (y-b) ^2 = r^2 donde (a, b) representa el centro de la circunferencia, y r es el radio. Esto implica que el acumulador tendrá tres dimensiones (a, b, r) en vez de dos como en las rectas, provocando además que haya que buscar entre todos los radios o en un intervalo especificado.

La transformada jugara un papel fundamental en nuestro proyecto ya que las trazas que encontremos en las muestras siempre serán circulares, por lo que una buena aplicación de esta técnica va a ser fundamental para poder extraer con precisión la información de estas.

Nosotros no implementaremos directamente esta técnica ya que la librería Opencv nos brinda una versión de esta, muy rápida, llamada HoughCircles.

## 6.2 Hough Circles

Los parámetros que escojamos para aplicar esta función no pueden ser triviales ya que el resultado podría ser erróneo. Para ello vamos a desgranar los parámetros que toma la función y exponer cual es el valor óptimo de cada uno de ellos.

RADIO MINIMO Y MAXIMO

El radio mínimo y máximo representaran los limites superior e inferior entre los que buscará circunferencias la función. Lo lógico es pensar que un rango amplio sería la opción más adecuada ya que a un mayor intervalo, más círculos detectará, pero esto no es así. Es cierto que, si la diferencia entre estos dos parámetros es muy pequeña, se perderán muchos círculos en cada imagen, pero en caso de que sea muy grande, los círculos serán detectados de forma imprecisa, dándonos un valor erróneo del radio.

No existe un valor universal para estos parámetros ya que dependerá del tamaño de las muestras, por lo que hemos implementado un mecanismo que seleccione al empezar la ejecución un valor óptimo. Este proceso consiste en calcular un radio medio aproximado utilizando la función que estamos viendo con parámetros muy generales. El radio medio que obtengamos será muy impreciso, pero nos servirá para ajustarlo en las próximas ejecuciones, estableciendo el radio mínimo, en el radio medio calculado menos 8 pixeles, y el radio máximo como el radio medio más 8 pixeles.

DISTANCIA ENTRE CENTROS

El segundo parámetro importante es la distancia mínima que deberá existir entre los centros de los círculos detectados. Al igual que los parámetros anteriores, no se puede usar un valor extremo ya que si fuese muy pequeño detectaría círculos superpuestos, y en el caso contrario, habría muchos círculos que no detectaría. El valor óptimo para este parámetro corresponderá a 2 veces el radio mínimo.

TAMAÑO DEL ACUMULADOR

Hemos visto en el aparatado teórico cómo funciona el acumulador en este tipo de operaciones. La función que estamos tratando nos deja elegir el tamaño se esté. El valor será el mismo en toda la ejecución y tendrá un valor de 50, cifra obtenida mediante la experimentación. En caso de aumentar este valor, habría muchos círculos correctos que no se detectarían, ya que solo detectaría los círculos en caso de que su valor en la matriz acumuladora fuese muy alto, pudiendo llegar a no detectar ningún circulo en alguna de las imágenes correctas; y si lo disminuyésemos ocurriría lo contrario. Dentro de la documentación este valor será tratado como “paraM2”.

Estos 3 parámetros serán los más importantes, pero no los únicos. El resto de ellos los dejaremos por su valor recomendado dentro de la documentación.

La llamada a la función tendrá el aspecto de la figura 13 y esta abstraída dentro de la función ‘obtenerCirculos’.

```
#Funcion que realiza la transformada sobre una imagen y devuelve los circulos encontrados
def obtenerCirculos(file, dir, tolerancia):
    #Abre la imagen
    image = cv2.imread(dir + "/" + file,0)
    try:
        #Busca los circulos con la transformada
        circles = cv2.HoughCircles(image, cv2.HOUGH_GRADIENT, 1,radioMinimo * 2,param1=50,param2=50,minRadius=radioMinimo,maxRadius=radioMaximo)
        #Redondea y transforma el resultado en un entero sin signo
        circles = np.uint16(np.around(circles))
        return circles
    except:
        return 0
```

La función HoughCircles devolverá null en caso de que no encuentre ningún circulo, por lo que nosotros tendremos que tratar una excepción de ‘AtributeError’ en la siguiente llamada a numpy.

## 6.3 Código

En general siempre que queramos extraer las trazas de una imagen, utilizaremos la función “obtenerCirculos” vista anteriormente. Esta función obtiene el radio mínimo y radio máximo de una variable global del programa, la cual habrá que calibrar antes de extraer los círculos.

Para realizar la calibración se extraerán los círculos utilizando como radio mínimo y máximo, 5 y 45 pixeles, respectivamente. Una vez se tienen los círculos, el programa los recorre, sumando su radio a una variable local y aumentando el contador en uno por cada uno. También aprovechara para extraer el píxel central de las trazas y sumarlo a otra variable.

```
for c in circulos[0,:]:
    try:
        #guardamos los datos que tenemos de la trazas analizada en tres variables.
        centroX = c[0]
        centroY = c[1]
        radio = c[2]
        #Y sumamos el radio a una variable local previamente inicializada
        sumaRadio += radio
        #Extraemos el pixel central de la imagen y se lo sumamos a una variable
        valorCentral = imagen[int(centroX), int(centroY)]
        valorTotalCirculo += valorCentral
        #aumnetamos el contador
        contadorUniversal += 1
```

Una vez hemos terminado de analizar una imagen, calculamos la media del valor RGB de su píxel central. Si este superior a un umbral de 200, la muestra se mueve a una carpeta de muestras blancas, y de lo contrario a una de muestras negras.

```
mediaCirculo = valorTotalCirculo / contadorCirculos
if mediaCirculo > valorUmbral:
    moverArchivo(f, dirMuestras, dirMuestrasBlancas)
else:
    moverArchivo(f, dirMuestras, dirMuestrasNegras)
```

El resto del programa sigue iterando sobre el resto de los archivos, sumando el radio de todas las trazas a la misma variable. Cuando terminamos de analizar todos los archivos calculamos la media del radio, le restamos y sumamos una variación para calcular el radio mínimo y máximo.

```
#Calculamos la media, y a partir de esta, el radio minimo y maximo
radioMedio = sumaRadio / contadorUniversal
radioMinimo = math.floor(radioMedio - variacionRadio)
radioMaximo = math.floor(radioMedio + variacionRadio)
print("radio medio: " + str(radioMedio))
#Si el radio minimo es negativo, lo dejamos en 0
if radioMinimo < 0:
    radioMinimo = 0
return radioMedio
```

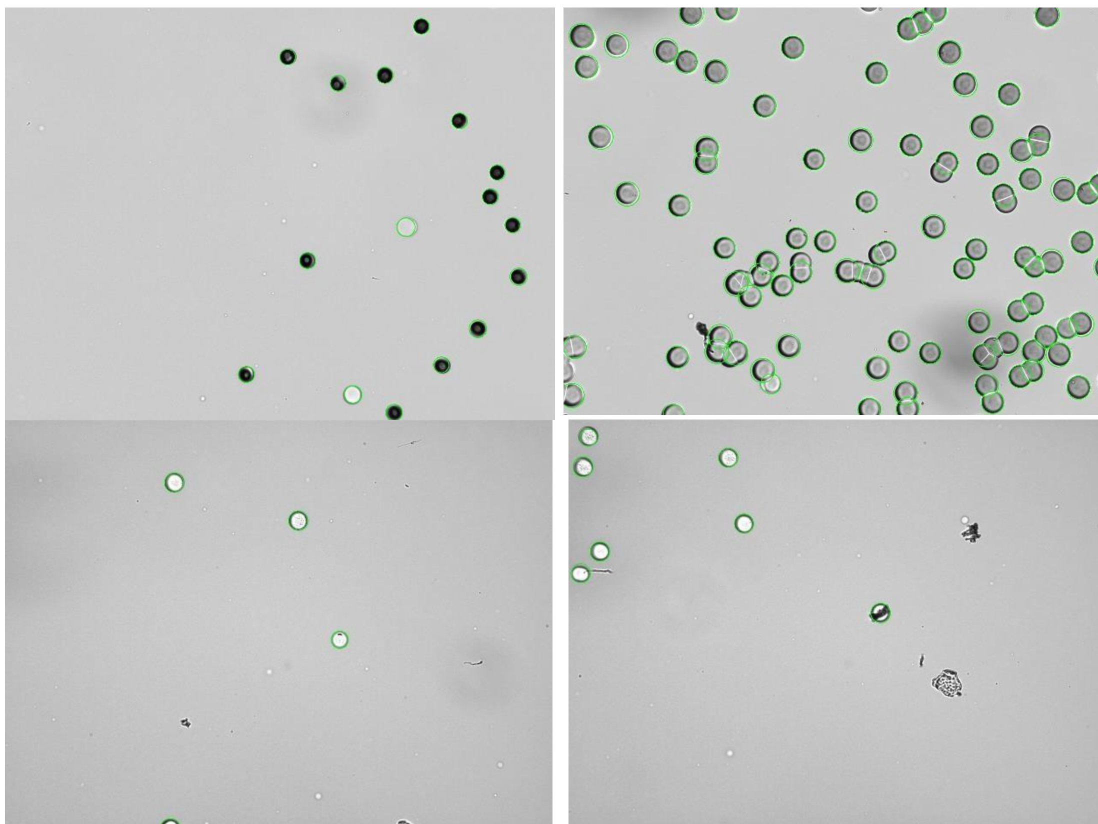
El parámetro de variación del radio vendrá predefinido por nosotros, y tendrá un valor de 8. Esto hará que las trazas que encontremos estén en un rango de 16 pixeles, lo que será suficiente para encontrar todas, ya que la mayoría tendrán el mismo radio.

En cuanto hemos terminado con la calibración, podremos extraer las trazas con total normalidad, y entrará en juego el software de apoyo.

## 6.4 Resultados

A diferencia del apartado de las redes neuronales, para medir los resultados de esta sección no dispondremos de métricas por lo que tendremos que evaluar la precisión de nuestro software de otra manera. Para medirlos, vamos a aplicar la transformada a distintas imágenes, ajustando los parámetros de forma automática, y señalar con un círculo verde aquellas trazas que detectemos para comprobar que se extrae el radio de forma precisa.

Al hacer la prueba con distintos tipos de imágenes obtuvimos resultados idénticos a los de la figura 22. Como se puede observar el software funciona bien tanto para círculos negros y blancos y de distintos tamaños. También se ve algún error en la segunda imagen debido a la acumulación de círculos, pero dichos errores no deberían perturbar los resultados.



Figuras 21: Distintos ejemplos de la transformada de Hough aplicada a las muestras del experimento

## 6.5 Aplicación

La aplicación de la función anterior se verá dentro del software de apoyo.

El software de apoyo constituirá la segunda parte del clasificador, y entrará en juego una vez la red neuronal ha terminado su clasificación. Su función será extraer características de las imágenes que, debido al redimensionado que sufren durante su preprocesamiento, la red convolucional es incapaz de detectar.

El proceso será muy similar tanto para muestras blancas como negras. La principal diferencia serán algunos parámetros. Por ejemplo, el borde de las muestras blancas será negro y por tanto se buscará utilizando como referencia un valor RGB oscuro.

Lo primero será extraer las trazas utilizando la función HoughCircles. Una vez las tenemos, crearemos un bucle que iterará sobre ellas. Dentro del bucle comprobará el tamaño del borde. Para ello llamara a una función llamada bordeBlanco o bordeNegro, dependiendo del tipo de muestras.

```
circulos = obtenerCirculos(f, dirMuestrasBlancas, tolerancia)
#Recorre el array de circulos devuelto por la transformada
for circulo in circulos[0,:]:
    #Carga la imagen con la libreria PIL
    imagen = cargarImagen(f, dirMuestrasBlancas)
    #Comprueba si es blanco el circulo
    esBlanco = centroBlanco(imagen, circulo)
    sumaCirculos += 1
    if esBlanco:
        #Suma el tamaño del borde a una variable
        sumaBorde += bordeNegro(imagen, circulo)
        #Y el tamaño del radio
        sumaRadio += circulo[2]
        #Aumenta en 1 el contador local
        circulosCorrectos+= 1
    else:
        #Si no es blanco el circulo,
        #lo suma a una variable de circulos incorrectos
        circulosIncorrectos += 1
```

La función que comprueba el tamaño del borde recorre desde una distancia igual al radio menos 10 pixeles hasta el radio mas 5 pixeles, sumando todos aquellos pixeles que sean del color del borde, blanco en caso de los círculos negros y viceversa.

```
#Lugar donde comenzará a comprobar los pixeles
comienzoMuestreo = radio -10
#Lugar donde dejará de comprobar los pixeles
finalMuestreo = radio + 5
#Bucle que servira para sumar pixeles
for mov in range(comienzoMuestreo, finalMuestreo):
    #Guarda el valor RGB en una variable
    dato = imagen[int(centroX), int(centroY) - mov]
    #comprueba si ese pixel es superior a un valor umbral de 175
    #Si el pixel es menor que ese valor se considerará oscuro, y por tanto parte del borde
    if dato < oscuridadUmbral:
        sumaBordes = sumaBordes + 1
#Devolverá el tamaño del borde
return sumaBordes
```

Cuando ha analizado, todas las trazas de una muestra, calculará el borde medio por circulo, el radio medio y la diferencia del radio y el borde, es decir, el radio del circulo si le extrajésemos el borde. Si el tamaño del borde es mayor que esta diferencia, se considerará que la imagen es incorrecta y por tanto se moverá a la carpeta de imágenes malas.

```
#Calculamos el borde medio, radio medio y la diferencia
radioMedio = sumaRadio / circulosCorrectos
bordeMedio = sumaBorde / circulosCorrectos
diferenciaRadioBorde = radioMedio - bordeMedio
#Comprobamos que el borde medio no sea mayor.
if bordeMedio > diferenciaRadioBorde:
    moverArchivo(file, dir, dirBad)
else:
    moverArchivo(file, dir, dirGood)
```

# 7. Implementación Final

En este apartado vamos a encargarnos de juntar todo el software que hemos desarrollado y explicar cómo funciona paso por paso.

Para empezar a ejecutar el programa necesitaremos disponer de aquellas muestras que queramos clasificar. Las deberemos colocar en un directorio dentro de la carpeta donde almacenemos el software, y entonces iniciaremos el programa.

La primera tarea que realiza el programa es crear aquellos directorios necesarios para el funcionamiento de este. Entre estos directorios destacamos uno que se llame 'Muestras Blancas' para almacenar las muestras blancas; y otro que se llame 'Muestras Negras', para almacenar el resto. Además, crea un directorio 'Resultados', con 3 subdirectorios: Descartes, Selección y Final. En el primero almacenarán aquellas imágenes que descartemos, el segundo la selección de imágenes correctas, y en el directorio final, loas imágenes ya procesadas y el resto de los resultados.

Entonces el programa comenzará a separar las imágenes en blancas y negras. Para ello detectará las trazas de las muestras con la transformada, y examinará los pixeles centrales de estas con la librería PILLOW. Para valorar el color de una muestra, hará la media de los valores de sus pixeles centrales. En caso de que sean mayores a un valor umbral, será una muestra blanca, y de lo contrario, será una muestra negra. A partir de aquí se empezará analizando las muestras blancas. En caso de que se terminase la clasificación y no hubiera muestras correctas, se empezaría de nuevo desde este punto, analizando las muestras negras en su lugar. Durante esta parte también se calibrarán los parámetros de la transformada de Hough.

Una vez se han separado las imágenes, haremos uso de la red neuronal. Para ello, cogemos el modelo ya entrenado, y con el método 'predict' de Keras, predecimos la clase de cada una de las muestras como hemos visto es el apartado de aplicación de la red neuronal.

Cuando se termina de utilizar la red neuronal, volvemos a emplear la transformada de Hough para comprobar el tamaño del borde tal y como hemos visto en el apartado de aplicación de computer visión, y así terminar con la clasificación de las imágenes.

La última fase del programa consiste en obtener la información de las muestras y realizar una representación gráfica de las mismas. La información que nos interesa es el número de trazas detectadas y sus radios, y a partir de estos datos también podemos obtener el radio medio y el error. Toda esta información la extraeremos con la transformada, y con el módulo pyplot de matplotlib crearemos un histograma para representarla.

## 7.1 Resultados

El output de la ejecución de este programa constará de varias partes. Lo primero que obtendremos será una carpeta con nuestra selección de imágenes sin procesar. Será útil tenerlas sin procesar para que, en caso de que haya alguna imagen clasificada incorrectamente que pudiese contaminar los resultados, se pueda eliminar y obtener rápidamente una selección más precisa.

La segunda parte del output consistirá en esas mismas imágenes, pero ya procesadas. Sabremos que se han extraído las trazas correctamente porque estas estarán marcadas por una circunferencia verde como en la figura 21. Además, por cada imagen obtendremos una tabla, como la de la figura 22, formada por 3 columnas: Su posición (en formato de coordenadas x y) y su radio.

TXT file for image-023

| x | y | radius |
|---|---|---|
| 1362 | 42 | 19 |
| 1378 | 188 | 19 |
| 1792 | 404 | 14 |
| 1676 | 200 | 20 |
| 1544 | 98 | 19 |
| 1878 | 268 | 14 |
| 1940 | 172 | 16 |
| 1268 | 108 | 19 |
| 1610 | 80 | 14 |

Figura 22: Output del software. Cada fila indica una nueva traza, dando como información su posición (x, y) y su radio.

Por último, el programa nos devuelve una imagen como la de la figura 23.

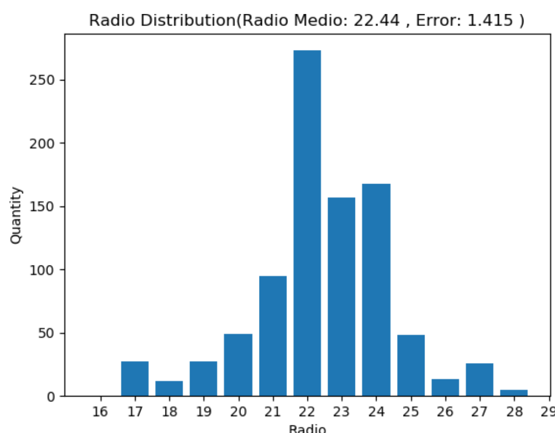



Figura 23: Histograma con la distribución de los radios de las trazas

Los histogramas además contendrán información sobre el radio medio y el error de la distribución.

Este elemento será el que más interés tenga ya que contiene la información que más tarde nos servirá para relacionar con las condiciones del experimento. Lo habitual y esperable es encontrarnos distribuciones muy parecidas a la de la Figura 25, ya que eso nos indicara que el radio de las trazas es similar para todas, y podemos achacar todas aquellas que se desvíen a partículas que no son protones o a una mala detección,

errores asumibles al no contaminar los resultados. Aun así, esta distribución puede variar y provocar que nos encontremos representaciones como la de la figura 24.

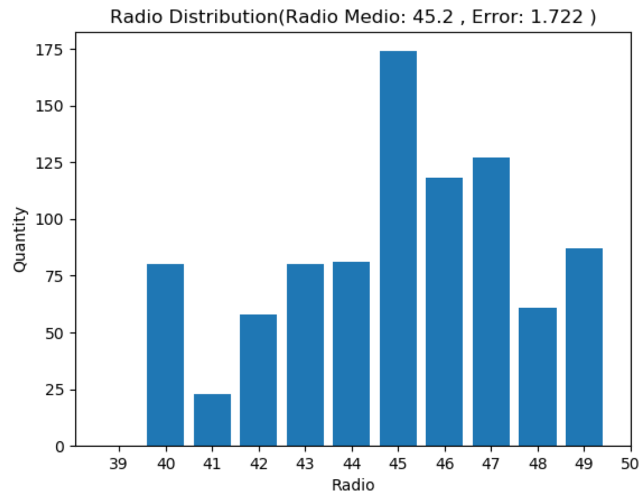



Figura 24: Histograma de radios con una distribución inusual.

El hecho de que los radios varíen tanto, y que el radio principal no sea tan claro, se debe a las condiciones del experimento.

Una vez tenemos todos los radios medios de una serie podemos dibujar un gráfico relacionándolos con la energía utilizada. Los resultados tendrán un aspecto como el de la Figura 25.

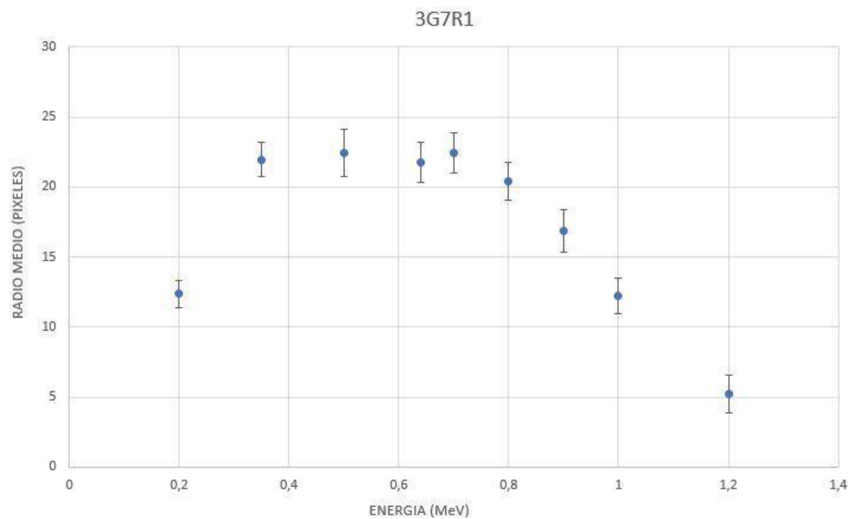



Figura 25: Dos diagramas donde relacionamos el radio medio de las trazas con la energía empleada en la aceleración.

En las imágenes se puede ver claramente como va evolucionando el tamaño de las trazas con distintos niveles de energía, con barras verticales de error, dando así por concluido el proyecto.

# 8. Conclusiones

En conclusión, el desarrollo de este proyecto ha dejado patente el gran potencial de la inteligencia artificial. Durante este tiempo, hemos empleado técnicas de diversos subcampos de este primero para clasificar y extraer información de imágenes microscópicas, mostrando como una tarea sencilla para los humanos, se convierte en un complejo sistema cuando intentamos darle dicha habilidad a un computador.

Como resultado del trabajo, hemos conseguido desarrollar una pieza de software capaz de clasificar imágenes microscópicas y extraer su información de forma automática y precisa, y poder crear una serie de representaciones para observar gráficamente como evoluciona el radio de las muestras para distintos niveles de energía.

## 8.1 Valoración Personal

La realización de este trabajo ha sido todo un reto en mi vida académica. No solo porque la tecnología utilizada es muy compleja sino, también porque la temática del proyecto se encontraba fuera de mi especialidad.

Todo este proyecto me ha servido para aprender mucho sobre inteligencia artificial, Deep Learning, y visión por computador, lo que puede ser muy útil de cara al futuro, y como valoración de mi trabajo considero que he sabido sobreponerme a todos los retos que se han presentado en este.

Desde que me asignaron el proyecto he tardado 6 meses en terminarlo, y aun así considero que con el tiempo necesario podría haber aplicado tecnologías más complejas.

## 8.2 Futuras Mejoras

La última sección de este documento la reservo para comentar posibles mejoras que se podrían realizar en el futuro. Todas estas actualizaciones irán orientadas a mejorar el rendimiento de la red de forma que se dependa más de esta y menos del software de apoyo.

La primera mejora consistirá en modificar el dataset de forma regular. A medida que se realice más veces el experimento, aumentará el número de muestras de las que dispongamos, y por lo tanto podremos entrenar más la red para mejorar su precisión.

También valoramos la posibilidad de implementar otros tipos de arquitecturas, o tecnologías más avanzadas como las Fast-RCNN [24] que forman parte de proyectos como YOLO [25].

Fast-RCNN es un tipo de red convolucional basada en regiones capaz de encontrar objetos en tiempo real dentro de una imagen o video. Sería interesante intenta aplicar este software a la detección de trazas en las muestras, y así conseguir desechar también la transformada Hough de nuestro software.

Para evitar tener que redimensionar las imágenes, se podría intentar aplicar convoluciones parciales sobre estas, para así poder trabajar con la resolución nativa.

La última mejora que se podría aplicar sería traducir el código a c++. Si el proyecto no lo he realizado en dicho lenguaje, ha sido porque no estoy familiarizado con la sintaxis, lo que podría haber sido una carga al final, pero todas las librerías que he utilizado se encuentran disponibles en dicho lenguaje, y podría mejorar mucho el rendimiento.

# Referencias

[1] SAS, What is artificial intelligence, Available at: https://www.sas.com/es_es/insights/analytics/what-is-artificial-intelligence.html

[2] M. Seimetz, P. Bellido, P. García, P. Mur, A. Iborra, A. Soriano, T. Hülber, J. García López,

M. C. Jiménez-Ramos, R. Lera, A. Ruiz-de la Cruz, I. Sánchez, R. Zaffino, L. Roso, and J. M. Benlloch,

Spectral characterization of laser-accelerated protons with CR-39 nuclear track detector

Available at: https://doi.org/10.1063/1.5009587

[3] Python, Available at: https://docs.python.org/3/tutorial/

[4] Anaconda, Available at: https://www.anaconda.com/

[5] TensorFlow, Available at: https://www.tensorflow.org/

[6] Scikit Learn, Available at: https://scikit-learn.org/stable/

[7] Keras, Available at: https://keras.io/

[8] Opencv, Available at: https://opencv.org/

[9] Pillow, Available at: https://pillow.readthedocs.io/en/stable/

[10] Numpy, Available at: https://www.numpy.org/

[11] Matplotlib, Available at: https://matplotlib.org/

[12] Skymind, A beginner's guide to neural networks, Available at: https://skymind.ai/wiki/neural-network

[13] Akshay Chandra Lagandula, Perceptron: the artificial neuron, Available at: https://towardsdatascience.com/perceptron-the-artificial-neuron-4d8c70d5cc8d

[14] Sagar Sharma, Activation functions in neural networks, Available at: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[15] Sumit Saha, A Comprehensive Guide to Convolutional Neural Networks, Available at: https://towardsdatascience.com/a-comprehensive-guide-toconvolutional-neural-networks-the-eli5-way-3BD2B1164a53

[16] Jason Brownlee, A gentle introduction to pooling layers, Available at: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

[17] SuperDataScience Team, Convolutional Neural Networks: Flattening, Available at: https://www.superdatascience.com/convolutional-neural-networks-cnn-step-3-flattening/

[18] Ravindra Parmar, Common loss functions, Available at: https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4D23

[19] Sirena, Optimizers for Training Neural Networks, Available at: https://medium.com/datadriveninvestor/optimizers-for-training-neural-networks-e0196662E21e

[20] Anas Al-Masri, What are Overfitting and Underfitting in Machine Learning?, Available at: https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690

[21] princeton.edu, Machine learning in Computer Vision, Available at: https://www.cs.princeton.edu/courses/archive/spring07/cos424/lectures/li-guest-lecture.pdf

[22] Divyansh Dwivedi, Face Detection for Beginners, Available at: https://towardsdatascience.com/face-detection-for-beginners-e58e8f21AAd9

[23] ed.ac.uk, Hough transform, Available at: http://homepages.inf.ed.ac.uk/rbf/HIPR2/Hough.htm

[24] Rohith Gandhi, Object Detection Algorithms, Available at: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

[25] YOLO: Real Timer Object Detection, Available at: https://pjreddie.com/darknet/yolo/

# A. Implementación y Funcionamiento del Código

A lo largo de este documento, hemos visto pequeños fragmentos de código que realizaban funciones especificas dentro del programa. En esta última sección se intentará exponer brevemente el código al completo, comentar algunas de sus peculiaridades y como se inicia el mismo.

El código se divide en dos secciones, la que utilizamos para crear la red neuronal y el software que clasifica las muestras y extrae la información, y los ficheros que las contienen son “RedNeuronal.py” y “CR39.py”.

Mi objetivo durante el desarrollo fue que el código fuera lo más legible posible intentando abstraer las funciones más confusas con otras funciones con nombres específicos de su función. Por ejemplo, cuando queremos cargar la imagen con la librería PIL, utilizamos una llamada como la siguiente:

```
foto = Image.open(dir + "/" + file)
imagen = foto.load()
```

Un lector corriente sin experiencia con Python puede no saber que estas líneas realizan dicha función, por lo que la encapsulamos del siguiente modo:

```
def cargarImagen(file, dir):
    foto = Image.open(dir + "/" + file)
    imagen = foto.load()
    return imagen
```

De esta forma cada vez que el lector vea una llamada a la función ‘cargarImagen’ sepa que está ocurriendo en ese punto del programa.

Otra peculiaridad del código de CR39.py es que está dividido en cinco fragmentos, llamados fases, para facilitar el debug, los cuales ejecutará la función main de forma secuencial.

Antes de iniciar el programa necesitarás tener instaladas todas las librerías para que este funcione y para ello utilizaremos pip desde la terminal. La llamada que tenemos que hacer es “pip install <nombre de la librería>” , siendo las librerías Tensor Flow, Keras, Opencv, PIL, Numpy, Matplotlib y pandas

Para iniciar el programa, bastará con situar en el directorio en el que encuentra el fichero con el código, una carpeta llamada “Muestras”, con las muestras dentro de élla. Una vez tenemos esto podremos iniciarlo dando doble clic y cuando haya terminado la ejecución encontraremos los resultados dentro de la carpeta “Results”.

Código RedNeuronal.py

```
import os
import cv2

import numpy as np
import matplotlib.pyplot as plt

from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split

from keras import backend as k
k.set_image_dim_ordering('tf')
from time import time

from keras.applications import vgg16

from keras.applications import resnet50
from keras.utils import np_utils
from keras.models import Sequential, Model, load_model
from keras.layers.core import Dense, Dropout, Activation,
Flatten
from keras.layers.convolutional import Convolution2D,
MaxPooling2D
from keras.optimizers import SGD, RMSprop, adam
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.python.keras.callbacks import TensorBoard

#os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'


img_row =128
img_col =128
num_channel = 1

img_data_list = []
labels_list = []
data_dir_list=["0", "1","2","3"]
#Recorre las carpetas del dataset
for dataset in data_dir_list:
    img_list = os.listdir("./" + dataset)
     #Prepara las imagenes
    for img in img_list:
        input_img = cv2.imread("./" + dataset + "/" + img)
        input_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
        input_img_resize = cv2.resize(input_img, (128,128))
        img_data_list.append(input_img_resize )
        labels_list.append(dataset)
```

```
img_data = np.array(img_data_list)
img_data = img_data.astype('float')
#nomaliza los valores
img_data /= 255
labels = np.array(labels_list)
#convierte las categorias numericas en vectores; ejemplo 1--
>[1,0,0,0], 2-->[0,1,0,0]
Y = np_utils.to_categorical(labels, 4)
#Expande las dimensiones para indicar el numero de canales
img_data = np.expand_dims(img_data, axis = 4)
#mezcla las iamgenes
x,y = shuffle(img_data,Y, random_state=2)
#crear los grupos de entrenamiento y testeo
X_train, X_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=2)

#Crea la red neuronal
model = Sequential()

#Capa convolucional
model.add(Convolution2D(32,(3,3), border_mode =
"same",input_shape = (128,128,1) ))
#Funcion ReLU
model.add(Activation("relu"))
#Capa convolucional
model.add(Convolution2D(32,3,3))
#Funcion ReLU
model.add(Activation("relu"))
#Funcion de Max Pooling
model.add(MaxPooling2D(pool_size = (2, 2)))
#Dropout de 0.5
model.add(Dropout(0.5))
#Flattening del input
model.add(Flatten())
#Capa fully connected
model.add(Dense(128))
#Funcion ReLU
model.add(Activation("relu"))
#Dropout de 0.5
model.add(Dropout(0.5))
#Capa fully connected
model.add(Dense(units = 4))
#Funcion Softmax y salida
model.add(Activation("softmax"))

#Iniciamos tensorboard para poder ver las estadísticas luego
tensorboard = TensorBoard(log_dir="logs/{}".format(time()))
#Compilamos
```

```
model.compile(loss="categorical_crossentropy",
optimizer="adam",metrics=["accuracy"])
#Entrenamos la red
model.fit(X_train,y_train,batch_size=32,epochs=20,verbose=1,vali
dation_data=(X_test,y_test),callbacks=[tensorboard])
```

Código CR39.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jun  4 10:55:05 2019

@author: jacagi
"""

import os
import keras
import numpy as np
import cv2
from PIL import Image
from keras.preprocessing.image import img_to_array
import matplotlib.pyplot as plt
import pandas
import math
import matplotlib.mlab as mlab
import csv

#Variables globales en las que se especifican rutas o parámetros
dirMuestras = "./Muestras"
dirMuestrasBlancas = dirMuestras + "/Blancas"
dirMuestrasNegras = dirMuestras + "/Negras"
dirResults = "./Results"
dirGood = dirResults + "/Seleccion"
dirBad = dirResults + "/Descartes"
dirFinal = dirResults + "/Final"
dirModels = "./Models"
directorios = [dirResults ,dirGood ,dirBad ,dirFinal,
dirMuestrasBlancas,  dirMuestrasNegras]
opcionFicheros = "f"
opcionDirectorios = "d"
opcionBlancas = "W"
opcionNegras = "B"
radioMinimo = 10
radioMaximo = 35
minimoCirculosCorrectos = 5
constanteCirculosIncorrectos = 1.5
resultsCSV = "results.csv"
ficheroResultados = "results.txt"
nombreGrafica = "histogram.png"
```

```
nombreModelo = "MyModel.h5"
maximoCirculosProcesados = 100

#Función main del programa
def main():
	print("Creando Directorios")
	createDirs()
	detectarTipoCirculos()
	print("Empezando Primera Fase")
	PrimeraFase(opcionBlancas)
	print("Empezando Segunda Fase")
	SegundaFaseBlanca()
	#Si después de la clasificación de muestras blancas,
	#no hay un mínimo de 5, se procede a analizar las muestras
negras
	act = obtenerArchivos(dirGood, opcionFicheros)
	if len(act) < 5:
		PrimeraFase(opcionNegras)
		SegundaFaseNegra()
	print("Empezando Tercera Fase")
	TerceraFase()
	print("Empezando Cuarta Fase")
	CuartaFase()
	print("Fin")

#Durante esta fase se utiliza la red para clasificar las
muestras
def PrimeraFase(Opcion):

	currentModel = keras.models.load_model(dirModels + "/" +
nombreModelo)
	dataPrepared = prepareData(Opcion)
	data = dataPrepared[0]
	fileNames = dataPrepared[1]
	predictions = currentModel.predict(data)
	processPredictions(predictions, fileNames, Opcion)

#La segunda fase será distinta para los dos tipos de muestras
#Durante esta parte, se analizarán los bordes y color de las
trazas
def SegundaFaseBlanca():
	tolerancia = 30
	files = obtenerArchivos(dirMuestrasBlancas, opcionFicheros)
	for f in files:
		circulosIncorrectos = 0
		circulosCorrectos = 0
		sumaRadio = 0
		sumaBorde = 0
```

```
        sumaCirculos = 0
        try:
            circulos = obtenerCirculos(f,
dirMuestrasBlancas, tolerancia)
            for circulo in circulos[0,:]:
                #Establecemos un máximo de círculos a
procesar para evitar que el tiempo de ejecución sea muy alto
                if sumaCirculos <
maximoCirculosProcesados:
                    try:
                        imagen = cargarImagen(f,
dirMuestrasBlancas)
                        esBlanco = centroBlanco(imagen,
circulo)
                        sumaCirculos += 1
                        if esBlanco:
                            sumaBorde +=
bordeNegro(imagen, circulo)
                            sumaRadio += circulo[2]
                            circulosCorrectos += 1
                        else:
                            circulosIncorrectos += 1
                    except:
                        continue

            validacionImagenBlanca(f, dirMuestrasBlancas,
circulosCorrectos, circulosIncorrectos, sumaRadio, sumaBorde)
        except:
            moverArchivo(f, dirMuestrasBlancas, dirBad)

def SegundaFaseNegra():
    tolerancia = 30
    files = obtenerArchivos(dirMuestrasNegras, opcionFicheros)
    for f in files:
        circulosIncorrectos = 0
        circulosCorrectos = 0
        sumaRadio = 0
        sumaBorde = 0
        sumaCirculos = 0
        try:
            circulos = obtenerCirculos(f, dirMuestrasNegras,
tolerancia)
            for circulo in circulos[0,:]:
                if sumaCirculos <
maximoCirculosProcesados:
                    try:
                        imagen = cargarImagen(f,
dirMuestrasNegras)
```

```
                                esNegro = centroNegro(imagen, 
circulo)
                                sumaCirculos += 1
                                if esNegro:
                                     sumaBorde += 
bordeBlanco(imagen, circulo)
                                     sumaRadio += circulo[2]
                                     circulosCorrectos += 1
                                else:
                                     circulosIncorrectos += 1
                           except:
                                continue

                validacionImagenNegra(f, dirMuestrasNegras, 
circulosCorrectos, circulosIncorrectos, sumaRadio, sumaBorde)
           except:
                moverArchivo(f, dirMuestrasNegras, dirBad)

#Esta fase consistirá en extraer la información de las trazas, y 
guardarla en un fichero csv
def TerceraFase():
     tolerancia = 25
     erroresUmbral = 6
     initCSV()
     files = obtenerArchivos(dirGood,opcionFicheros)
     for f in files:
          errores = 0
          circles = obtenerCirculos(f, dirGood, tolerancia)
          image = cv2.imread(dirGood + "/" + f,0)
          cimg = cv2.cvtColor(image,cv2.COLOR_GRAY2BGR)
          for c in circles[0,:]:
               cimgProvi = dibujarCirculo(f, dirGood, cimg, c)
               if cimgProvi is None:
                    errores = errores + 1
               else:
                    cimg = cimgProvi
                    writeCSV(f, c)
          if errores < erroresUmbral:
               cv2.imwrite(dirFinal + "/" + f,cimg)
          else:
               print(f)
               print("Too mmuch errors")
               moverArchivo(f, dirGood, dirBad)

#Esta última fase creará gráficas y calculará el radio medio a 
partir de la información extraída en la fase anterior.
def CuartaFase():
```

```
      radioMedio = calcularRadioMedio()
      error = calcularError(radioRow, radioMedio, totalCirculos)

      file = open(dirFinal + "/" + ficheroResultados,"w+")
      file.write("Average Radio: " + str(radioMedio) + "\n")
      file.write("Error: " + str(error) + "\n")
      file.write("Total Circles: " + str(totalCirculos) + "\n")
      file.close()

      crearGrafica(radioMedio, error)

#Primera parte del programa donde se separarán las muestras
según su color y se calculará el radio medio
def detectarTipoCirculos():
      tolerancia = 40
      valorUmbral = 175
      radioMinimo = 0
      radioMaximo = 45
      variacionRadio = 8
      ficheros = obtenerArchivos(dirMuestras, opcionFicheros)
      sumaRadio = 0
      contadorUniversal = 0
      for f in ficheros:
            circulos = obtenerCirculos(f, dirMuestras,
tolerancia)
            imagen = cargarImagen(f, dirMuestras)
            valorTotalCirculo = 0
            contadorCirculos = 0
            try:
                  for c in circulos[0,:]:
                        try:
                              centroX = c[0]
                              centroY = c[1]
                              radio = c[2]
                              valorCentral = imagen[int(centroX),
int(centroY)]
                              sumaRadio += radio
                              contadorUniversal += 1
                              valorTotalCirculo += valorCentral
                              contadorCirculos += 1
                        except:
                              continue
                  mediaCirculo = valorTotalCirculo /
contadorCirculos
                  if mediaCirculo > valorUmbral:
                        moverArchivo(f, dirMuestras,
dirMuestrasBlancas)
                  else:
```

```
                        moverArchivo(f, dirMuestras, 
dirMuestrasNegras)
            except:
                  moverArchivo(f, dirMuestras, dirBad)

      radioMedio = sumaRadio / contadorUniversal
      radioMinimo = radioMedio - variacionRadio
      radioMaximo = radioMedio + variacionRadio
      print("radio medio: " + str(radioMedio))
      if radioMinimo < 0:
            radioMinimo = 0

#Método que se encarga de crear el histograma con la 
distribución de los radios.
def crearGrafica(radioMedio, error):
      columnaDatos = "Radio"
      datos = readCSV(resultsCSV, dirFinal, columnaDatos)
      aux = []
      datos2 = []
      radioMedio = round(radioMedio, 2)
      for d in datos:
            if d > radioMedio - 6 and d < radioMedio + 6:
                  datos2.append(d)

      for i in range(min(datos2)-1, max(datos2) + 2):
            aux.append(i)
      plt.hist(datos2,bins = aux,rwidth = 0.8, align='left')
      plt.xticks(aux,aux)
      plt.xlabel('Radio')
      plt.ylabel('Quantity')
      plt.title('Radio Distribution(Radio Medio: ' + 
str(radioMedio) + ' , Error: ' + str(error) +' )')
      plt.savefig(dirFinal + "/" + nombreGrafica)

#Función que calcula el radio medio
def calcularRadioMedio():
      columna = "Radio"
      radioRow = readCSV(resultsCSV, dirFinal, columna)
      a = []
      for r in radioRow:
            a.append(r)

      counter = Counter(a)
      mostCommon = counter.most_common(1)
      elem = mostCommon[0][0]
      sum = 0
      cont = 0
      for i in a:
```

```
            if i in range(elem - 3, elem + 4):
                  sum += i
                  cont += 1
      radioM = sum / cont
      print(radioM)
      return radioM

#Función que calcula el error
def calcularError(radioMedio):
      columna = "Radio"
      radioRow = readCSV(resultsCSV, dirFinal, columna)
      a = []
      for r in radioRow:
            a.append(r)

      counter = Counter(a)
      mostCommon = counter.most_common(1)
      elem = mostCommon[0][0]

      error = 0
      cont = 0
      for i in a:
            if i in range(elem - 3, elem + 4):
                  error = error + ((i - radioMedio)**2)
                  cont = cont + 1

      error = error / (cont - 1)
      error = math.sqrt(error)
      error = round(error, 3)
      print(error)
      return error

#Función que se encarga de dibujar los círculos sobre la imagen
que recibe
def dibujarCirculo(file, dir, cimg, info):
      centroX = info[0]
      centroY = info[1]
      radio = info[2]
      imagen = cargarImagen(file, dir)

      cv2.circle(cimg, (centroX, centroY), radio,(0, 255, 0), 2)
      cv2.circle(cimg, (centroX, centroY), 2, (0, 0, 255), 3)
      return cimg

#Función que utilizamos para escribir sobre el fichero csv la
información de los círculos
def writeCSV(file, info):
      centroX = info[0]
```

```
	centroY = info[1]
	radio = info[2]
	name, extension = file.split(".")
	with open(dirFinal + "/" + resultsCSV, mode='a') as 
circles_file:
		circles_writer = csv.writer(circles_file, 
delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)
		circles_writer.writerow([name, centroX, centroY, 
radio])


#Función para inicializar el fichero csv
def initCSV():
	with open(dirFinal + "/" + resultsCSV, mode='w') as 
circles_file:
		circles_writer = csv.writer(circles_file, 
delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)
		circles_writer.writerow(['File Name', 'x', 'y', 
'Radio'])


#Función para validar las imágenes negras una vez hemos extraído 
el borde
#Estas no serán correctas si la relación del radio respecto al 
borde es superior a 3
#Además se comprueba que haya un mínimo de círculos correctos
def validacionImagenNegra(file, dir, circulosCorrectos, 
circulosIncorrectos, sumaRadio, sumaBorde):
	if circulosCorrectos > 0:
		maximoCirculosIncorrectos = circulosCorrectos / 
constanteCirculosIncorrectos
		radioMedio = sumaRadio / circulosCorrectos
		bordeMedio = sumaBorde / circulosCorrectos
		relacionRadioBorde = radioMedio / bordeMedio
		print(" ----- ")
		print("Borde: " + str(sumaBorde))
		print("Radio: " + str(sumaRadio))
		print("Borde: " + str(bordeMedio))
		print("Radio: " + str(radioMedio))
		print("Relacion: " + str(relacionRadioBorde))
		print(" ----- ")
	if circulosCorrectos > minimoCirculosCorrectos and 
circulosIncorrectos < maximoCirculosIncorrectos and 
relacionRadioBorde > 3:
		moverArchivo(file, dir, dirGood)
	else:
		moverArchivo(file, dir, dirBad)
```

```
#Función para validar las imágenes negras una vez hemos extraído el borde
#Estas no serán válidas si la diferencias entre el radio y el borde multiplicado por un factor de Reducción para mejorar su funcionamiento
#es menor que el borde medio
#Además se comprueba que haya un mínimo de círculos correctos
def validacionImagenBlanca(file, dir, circulosCorrectos, circulosIncorrectos, sumaRadio, sumaBorde):
	if circulosCorrectos > 0:
		maximoCirculosIncorrectos = circulosCorrectos / constanteCirculosIncorrectos
		radioMedio = sumaRadio / circulosCorrectos
		bordeMedio = sumaBorde / circulosCorrectos
		diferenciaRadioBorde = radioMedio - bordeMedio
		factorDeReduccion = 0.75
	if circulosCorrectos > minimoCirculosCorrectos and circulosIncorrectos < maximoCirculosIncorrectos and bordeMedio < diferenciaRadioBorde * factorDeReduccion:
		moverArchivo(file, dir, dirGood)
	else:
		moverArchivo(file, dir, dirBad)

#Función que comprueba que el centro de una muestra sea blanco
def centroBlanco(imagen, info):
	centroX = info[0]
	centroY = info[1]
	oscuridadUmbral = 180
	if imagen[int(centroX), int(centroY)] < oscuridadUmbral:
		return False
	else:
		return True

#Función que comprueba que el centro de una muestra sea negro
def centroNegro(imagen, info):
	centroX = info[0]
	centroY = info[1]
	oscuridadUmbral = 200
	if imagen[int(centroX), int(centroY)] > oscuridadUmbral:
		return False
	else:
		return True
#Función que comprueba el borde para las muestras negras
def bordeBlanco(imagen, info):
	centroX = info[0]
	centroY = info[1]
	radio = info[2]
	sumaBordes = 0
```

```
    oscuridadUmbral = 240
    comienzoMuestreo = radio -5
    finalMuestreo = radio + 10
    for desvio in range(comienzoMuestreo, finalMuestreo):
        dato1 = imagen[int(centroX), int(centroY) - desvio]
        dato2 = imagen[int(centroX), int(centroY) + desvio]
        dato3 = imagen[int(centroX) - desvio, int(centroY)]
        dato4 = imagen[int(centroX) + desvio, int(centroY)]
        if dato1 > oscuridadUmbral:
            sumaBordes = sumaBordes + 1
        if dato2 > oscuridadUmbral:
            sumaBordes = sumaBordes + 1
        if dato3 > oscuridadUmbral:
            sumaBordes = sumaBordes + 1
        if dato4 > oscuridadUmbral:
            sumaBordes = sumaBordes + 1
    return sumaBordes

#Función que comprueba el borde para las muestras blancas
def bordeNegro(imagen, info):
    centroX = info[0]
    centroY = info[1]
    radio = info[2]
    sumaBordes = 0
    oscuridadUmbral = 170
    comienzoMuestreo = radio -10
    finalMuestreo = radio + 20

    for desvio in range(comienzoMuestreo, finalMuestreo):
            dato = imagen[int(centroX), int(centroY) -
desvio]
            if dato < oscuridadUmbral:
                sumaBordes = sumaBordes + 1

    return sumaBordes

#Función que carga una imagen usando PIL
def cargarImagen(file, dir):
    foto = Image.open(dir + "/" + file)
    imagen = foto.load()
    return imagen

#Función que realiza la transformada sobre una imagen y devuelve
los círculos encontrados
def obtenerCirculos(file, dir, tolerancia):
    #Abre la imagen
    image = cv2.imread(dir + "/" + file,0)
    try:
```

```
            #Busca los círculos con la transformada
            circles = cv2.HoughCircles(image, cv2.HOUGH_GRADIENT,
1,radioMinimo *
2,param1=50,param2=50,minRadius=radioMinimo,maxRadius=radioMaxim
o)
            #Redondea y transforma el resultado en un entero sin
signo
            circles = np.uint16(np.around(circles))
            return circles
      except:
            return 0

#Función que procesa las predicciones de la red neuronal
def processPredictions(predictions, fileNames, opt):
      if opt == "W":
            dirTarget = dirMuestrasBlancas
      else:
            dirTarget = dirMuestrasNegras
      numberPredictions = len(predictions)

      for index in range(0, numberPredictions):
            currentPrediction = predictions[index]
            currentFile = fileNames[index]
            if currentPrediction[0] < currentPrediction[1] +
currentPrediction[2] + currentPrediction[3]:
                  moverArchivo(currentFile, dirTarget, dirBad)

#Prepara las imagenes para la red neuronal
def prepareData(opt):
      if opt == "W":
            dirTarget = dirMuestrasBlancas
      else:
            dirTarget = dirMuestrasNegras
      files = obtenerArchivos(dirTarget, opcionFicheros)
      data = []
      names = []
      res = []
      for f in files:
            name, extension = f.split(".")
            if extension == "png":
                  image = cv2.imread(dirTarget + "/" + f)
                  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
                  image = cv2.resize(image,(128,128))
                  image = img_to_array(image)
                  data.append(image)
                  names.append(f)

      data = np.array(data, dtype="float") / 255
```

```
      data = np.expand_dims(data, axis = 4)
      res = [data, names]
      return res

#Función utilizada para mover archivos
def moverArchivo(file, currentDir, destinationDir):
      os.rename(currentDir + "/" + file, destinationDir + "/" +
file)

#Función utilizada para obtener los archivos de un directorio
def obtenerArchivos(name, opt):
      for root, dirs, files in os.walk(name):
            if opt == "f":
                  return files
            else:
                  return dirs

#Función usada para crear los directorios al iniciar el programa

def createDirs():
      for d in directorios:
            if not os.path.exists(d):
                  os.mkdir(d)

if___name___== "_main_":
    main()
```